Spring 5-2016

Application of Parallel Computing to Optimize Studies of Critical Exponents in the One-Dimensional Sznajd Model

Joseph Garcia
University of Maine

APPLICATION OF PARALLEL COMPUTING TO OPTIMIZE STUDIES OF CRITICAL EXPONENTS IN THE ONE-DIMENSIONAL SZNAJD MODEL

by

Joseph Garcia

A Thesis Submitted in Partial Fulfillment
of the Requirements for a Degree with Honors
(Engineering Physics)

The Honors College

University of Maine

May 2016

Advisory Committee:

Samantha C. Jones, M.F.A., Honors Preceptor
Margaret O. Killinger, Ph.D., Rezendes Preceptor for the Arts, Associate Professor of Honors
Susan R. McKay, Ph.D., Professor of Physics and Director of the Maine Center for Research in STEM Education
Thomas E. Stone, Ph.D., Adjunct Professor of Physics
Yifeng Zhu, Ph.D., Professor of Electrical and Computer Engineering

Abstract

The Sznajd model (SM) is a one-dimensional voter-like model used to study consensus in systems where information flows outward from like-minded neighboring agents. Here, we introduce long-range interactions to the SM via the parameter p, where $p \to 1$ is the mean-field limit (MFL) and $p \to 0$ the one-dimensional limit (1DL). Using Monte Carlo simulations and finite size scaling analyses to characterize the exit probability for $p > 0$, we find a step function reliant on two p-dependent exponents. By examining the exponents' behavior in the 1DL, we comment on the functional form of the exit probability in one dimension—its nature has been an open question. Complimenting this limiting approach, we also simulate the 1D case via two parallelization techniques (task-level and data-level). Finally, we investigate the quantitative nature of consensus time and system magnetization across the p-spectrum.

We find that one of the exponents grows rapidly in the 1DL, its behavior suggesting divergence in this limit; the other stays approximately constant, although more low-p runs are needed to verify both values. Combined, these two exponents give rise to a functional form that well approximates a sigmoidal polynomial that almost exactly fits the original SM simulation results. We also find that consensus time at fixed system size is proportional to $p^{-1.1}$.

In testing the parallel code, we find that the task-level parallelism approach generates a speedup nearly equal to the number of processors applied; conversely, the data-level parallelism approach results tentatively suggest a superlinear speedup for constant system size.

Acknowledgements

When I look back on this thesis adventure, there are many people I can think of who made sure that each step I took—whether in data analysis, programming, or writing—was the right one to make. Without their help, I would surely be lost.

I would first like to thank Tom Stone and Susan McKay for being such excellent advisors these past few years. In addition to helping me understand what I was studying and how to go about getting the most from my simulations (as well as providing me some excellent resources with which to do those simulations) they have also taught me how to think, work, and communicate like a physicist.

I also extend my thanks to Yifeng Zhu, who helped me to make the most of the parallel code that I was writing, and Steve Cousins at the Advanced Computing Group, who helped me in finding my way on the powerful computer cluster he oversees. Additionally, I would like to thank Sam Jones and Mimi Killinger for their kind words, willingness to help, and advice that have kept me motivated to finish this thesis.

I would also like to send thanks to the University of Maine CUGR, the Department of Physics and Astronomy, and Dr. François Amar for their funding that allowed me to travel to the American Physical Society's March Meeting this year. At that meeting I was really able to crystallize the research that I have been working on these past few years into something cohesive and final.

Finally, I would like to thank my parents and my sisters, whose constant interest in my work kept me in a productive and driven mode, helping me to bring this thesis to completion.

Table of Contents

Table of Figures

Chapter 1 - Introduction

A Basic Explanation of The Sznajd Model

When considering a collection of individuals, all of which have the capability to influence each other's opinions, there is a natural inclination to consider the political nature of such a system. There has already been some significant activity in this area—for example, a modified version of the Sznajd Model (SM) [1, 2] that we use as a baseline model for this study was able to successfully replicate the election results for candidates in Brazilian state and federal deputy proportional elections about 10 years ago [3]. Indeed, the SM is not limited to large-scale systems, as the model proved quite robust in predicting proportional elections in the smaller political environment of cities, too (2000, Sao Paolo, Brazil, city councillor elections) [3].

In another fascinating (and quite relevant given we are in an election year) topic, we mention that the normally binary-state SM has also been altered by Stauffer to include a multi-party political structure [4, 5]. In his paper, he proposed a four-party system, where two parties are centrist and the other two are hard-line. In his modification of the SM each lattice site is initially either "empty, with probability 1/2, or has one of four possible opinions 1, 2, 3 or 4 (like in the Potts model), with probability 1/8 each. Then, at each time step every occupied site [agent] tries to move to an empty neighbor" [4]. He found that the moderate parties *always* won, but the extremist parties often maintained a small minority at the end. Given that compromise is usually what moves a society forward, yet hardliners continue to keep their less popular opinions, this is a pleasing and intuitive result.

As useful as it may be in modeling proportional elections, the SM is certainly not limited to the realm of politics. In fact, a stock-exchange oriented version of the original model was created to simulate the behavior of price formation in a stock market [6]. Although sharing the same "outflow" of opinion characteristic of its SM forebear, it included these two distinctions:

1. If two neighboring sites (agents) S_i and S_{i+1} disagree, the outside sites S_{i-1} and S_{i+2} randomly choose whether to buy or sell (the subscript i refers to position on a 1D lattice).
2. One fundamentalist (a person who has the knowledge to buy and sell appropriately according to the supply and demand of a given market) is included in each lattice.

Ultimately, they found this simple model to be an excellent first order approximation for the time trends of multiple real stocks.

Now that we have introduced some applications of the SM, we will proceed to discuss the mechanisms by which it works and what has been learned about it so far. To begin, the one-dimensional (1D) Sznajd Model was first put forth by Katerina Sznajd-Weron and Jozef Sznajd in their 2000 paper "Opinion evolution in closed community" [1]. In this model, a population of L individuals (sites) is modeled as a one-dimensional ring (lattice) where each site i can be in one of two states (opinions)—up (+1) or down (-1), represented by the variable $S_i = \pm 1$, in analogy with the Ising magnetic dipole moment variable. In the version of the SM we test in this study [2], given two sites i and $i+1$ with opinions S_i and S_{i+1} (where $i = 1, 2, 3, \ldots, L$) the rules for the update of site opinions are as follows:

1. Pick a random site S_i in the lattice; its neighbor is S_{i+1}, and they form what we will refer to as a site (or agent) pair.
2. If $S_iS_{i+1} = 1$ (two neighbors agree), then S_{i-1} and S_{i+2} are set equal to S_i . This is the idea of "strength in numbers," proposed in [1]. If $S_iS_{i+1} = -1$ (two neighbors disagree), then no sites change opinion.
3. Regardless of whether the site pair is in agreement or not, the system moves from time t_1 (at the pair selection) to t_2 (immediately after the attempted update) as follows: $t_2 = t_1 + 1/L$. Time is measured in Monte Carlo steps (MCS), where L pair selections is 1 MCS.
4. Repeat steps 1-3 until either up or down-state consensus is reached.

Before proceeding with a more in-depth look at the dynamics of the SM, we will discuss some bookkeeping terms. Henceforth, $S_iS_{i+1} = 1$ scenarios will be referred to as update possibility 1 (UP1), while $S_iS_{i+1} = -1$ scenarios will be referred to as update possibility 2 (UP2). Additionally, the time that it takes to reach consensus is called the consensus time, which we label as τ. How the consensus time varies with lattice topology, different dynamical rules, and other modifications to the model that seek to capture some particular aspect of real-life society is often studied extensively; additionally, what final state emerges most often is also fundamental to the SM.

We now move on to the physical realization of the SM initial conditions. For a given lattice of sites that needs to be updated to consensus, there is an initial percentage x that have the up opinion; $x = 0$ would mean that no sites are initially up, while $x = 1$ would mean that all sites are initially up. In these limiting cases, it is clear that that there would be no opinion evolution of the lattice. For all other cases ($0 < x < 1$), the initial

sites in the up state are distributed randomly on the ring, with no correlation in their positions.

The primary quantity of interest in the Sznajd Model is the exit probability $E(x,L)$, the probability that a lattice of size L will reach consensus in the up state given an initial fraction of sites x that have the up opinion. If a lattice is allowed to evolve from initial conditions to consensus (either up or down) N times (N trials) at a single x-value, and it reaches consensus in the up state U times, then

$$E(x) \stackrel{\text{def}}{=} \frac{U}{N} \#(1)$$

Based on the discussion to the present, the Sznajd Model lattices may seem like black boxes, where only the input (initial conditions) and output (consensus) are known. However, one quantity that can be used to gauge the overall behavior of a lattice during a trial is the magnetization m, defined as follows:

$$m(t) = \frac{N_{up}(t) - N_{down}(t)}{L}, \#(2)$$

where $N_{up}(t)$ and $N_{down}(t)$ are the number of sites with the up and down opinions at time t, respectively. By plotting m versus t, one can see the domination (or lack thereof) of a particular opinion throughout the lifetime of a lattice. If $m > 0$, more than 50% of the sites are up, while if $m < 0$, the majority are down; $m = 0$ corresponds to exactly half up and half down.

Until now, only short-range, nearest-neighbor interactions within the lattice have been discussed. In our study we introduce a long-range (LR) interaction probability p such that at each pair selection the individual sites of the pair (assuming they satisfy UP1) each have the chance to update one faraway site with probability p. We include this long

range interaction probability to bring greater realism to the lattices under study, as it is often the case that agents separated by distance can influence each others' opinions. With this factor included, we generalize $E(x,L)$ to become $E(x,p,L)$ and the SM becomes the LRSM.

To gain a more physical understanding of p, one can think of it as a society's faith in (or usage of) its available long range communication resources (mail, telephones, cellphones, the Internet, social media). If $p = 0$, people will not use (or do not have access to) devices that could be used to influence far-away associates; if $p = 1$, people will only communicate with each other via long-range communication devices (pretty common in today's world!). From here forward, we will refer to the $p = 0$ limit as the one-dimensional limit (1DL), while the $p = 1$ limit will be referred to as the mean-field limit (MFL).

With this perspective, one can easily see that a $p = 1$ society (lattice) should evolve its collective opinion very quickly. That is, it can reach consensus much more quickly than a $p = 0$ society. Indeed, previous studies have shown that a small amount of long-range interaction can significantly alter the dynamics of many systems [7].

An Explanation of the Topics of Interest

With an introduction to the basic behavior of the Sznajd Model, some of its applications, and the primary quantities used for its characterization now made, we can comment on what higher level analysis each data contribution allows us to do.

The parameters of the most fundamental interest in our study, though which we have not yet mentioned, are called the "critical exponents" which are, in this case, fitting parameters derived from the simulation exit probability data. In general, critical

exponents describe generic power-law behaviors characterizing transitions in a system. In a broader sense, a set of critical exponents can help characterize a system as part of a larger universality class. By considering what the universality class of a particular model/system is, researchers are able to map more complicated systems to simpler ones, assuming they are members of the same universality class. For binary-state models such as the SM, Biswas and Sen (B & S) came up with the following exponent-dependent equation to predict exit probability [8].

$$E(x,L) = \frac{1}{2}\left[tanh\left(\lambda\frac{x - x_c}{x_c}L^{1/\nu}\right) + 1\right] \#(3)$$

Here, L is the lattice size, λ and ν are the critical exponents, and $x_c = 0.5$ by the symmetry of the up and down opinions. B & S found a universal value of $\upsilon = 2.5 \pm 0.03$.

In our study, we postulate that Eq. (3) is correct for a given $p > 0$ and then determine υ and λ at that p. We simulate lattices of size L = 100, 500, 1000, 5000, 10000, 50000, and 100000, each of which is subjected to a range of p-values that includes p = {1.0, 0.75, 0.5, 0.25, 0.1, 0.075, 0.05, 0.025, 0.01, 0.0075, 0.005, 0.0025, 0.001, and 0.00}. As the intuitive explanation of p in the preceding section might indicate to the reader, simulations of any lattice size at $p = 0$ are very slow. By tuning $p \rightarrow 0$, then, we hope to extrapolate the values of $\upsilon(0)$ and $\lambda(0)$ from the limiting behavior of $\upsilon(p)$ and $\lambda(p)$ as p approaches 0, ideally saving time since we are not directly starting at the slowest possible case.

Aside from simply being fitting parameters, the real importance of $\upsilon(0)$ and $\lambda(0)$ can be appreciated once we consider the first known functional form of exit probability for the SM, postulated as an approximation in [2] via an application of the Kirkwood approximation and later expressed as exact in [9]. The equation is as follows:

$$E(x) = \frac{x^2}{x^2 + (1-x)^2} \quad \#(4)$$

The authors in [9] found Eq. (4), which we will occasionally refer to as the "sigmoidal polynomial," to be in tight statistical agreement with the simulation data. Its derivation, as illustrated in [2], is fairly simple, and is made possible by assuming that the requisite conditions for mean field theory (MFT) are satisfied, namely, that the net "feeling of opinion" at a given site is due to an averaging of the opinion influences being imposed upon it by each of its many neighbors.

Although this prediction worked, the MFT assumptions made are completely inconsistent with the reality of the 1DL; each site can only be influenced by the neighbors immediately adjacent to it, so there are only 2 sites to average an effect over. This is far beneath the threshold for MFT, so why these assumptions appear to work has been an open question. One explanation proposes that the opinion of the first agreeing pair selected is a primary factor in the form of Eq. (4) [9].

With this conflict in mind, we bring to bear the significance of $\upsilon(0)$ and $\lambda(0)$ by noting our goal: to determine just how well Eq. (3) and Eq. (4) agree in the 1DL. Indeed, the applicability of Eq. (4) to the 1DL can certainly not be denied. However, Eq. (3), fundamentally an ansatz which incorporates finite size scaling effects, is more of a mystery. Then, by letting $p \to 0$ and finding $\upsilon(0)$ and $\lambda(0)$ from the limiting behavior of $\upsilon(p)$ and $\lambda(p)$, one can see that we are trying to determine if the critical exponents at $p = 0$ for Eq. (3) generate an $E(x)$ that agrees with Eq. (4).

The equality of the expressions would mean that Eq. (4), though counterintuitive, might simply be a special case of Eq. (3). However, if the two equations do not converge in the 1DL, that brings up the possibility that the two are fundamentally incompatible—a

system with any possibility of long-range interactions does not undergo the same type of phase transition in its exit probability that a $p = 0$ system does.

To expand on our discussion of Eq. (3), we make a couple of notes about its behavior in some limiting cases. As $L \to \infty$ (the thermodynamic limit), $E(x,L)$ approaches a step function rising at x_c , while as $\lambda \to 0$, $E(x,L)$ becomes a strongly linear and increasingly flattening function still passing through the point (0.5,0.5); interestingly, this function never equals 1 within the range $0 \leq x \leq 1$). As $\upsilon \to \infty$, $E(x,L)$ becomes increasingly linear, still passing through (0.5,0.5); conversely, as $\upsilon \to 0$, $E(x,L)$ behaves exactly like it does in the limit of $\lambda \to \infty$ (becoming strongly steplike). As the product $\lambda L^{1/\nu} \to \infty$, (i.e., $\upsilon \to 0$ and/or $\lambda \to \infty$), Eq. (3) also becomes very step-like, squashing the asymptotes that it has at $E(x,L) = 0$ and $E(x,L) = 1$ (this is good, since $x = 0$ and $x = 1$ must have $E(x,L) = 0$, and $E(x,L) = 1$, respectively).

Critical exponents are used in describing the universality class of models, so it is interesting to note here that if $\upsilon(p)$ and $\lambda(p)$ are constant values, that would indicate a constancy in universality class for the LRSM across p (not the case for multiple models) [8, 10, 11]. This would mean that no matter what p-value a particular society (lattice) has, the nature of the phase transition with respect to the initial up-opinion density x would be the same, and Eq. (4) would hold for all p. In effect, adding a long-range interaction probability would not impact system consensus vs. x.

To give some meaning to what the graphical behavior of $E(x)$ means, we note that if it was strongly steplike (stepping at $x = 0.5$) , that would indicate a purely "majority-rules" society— that is, the lattice will always adopt the opinion of the initial majority, no matter what. Conversely, if $E(x,p,L)$ ended up being purely linear (i.e., $E(x,p,L) = x$) it

would be the seemingly more realistic alternative that a party's chance of victory was not a binary affair, but rather directly proportional to its initial presence in a system.

We go now to a less controversial facet of the SM and discuss its consensus time τ. There are many different ways to choose which parameters (that is, x, L or p) are fixed and which are varying in studying consensus time, but the following two configurations

- $\tau_1(\boldsymbol{x}=0.5, \boldsymbol{p}, L)$
- $\tau_2(\boldsymbol{x}=0.5, p, \boldsymbol{L})$

are the most commonly investigated. Here, the bolded-italic notation signifies that a quantity is being held constant. For example, in considering τ_1, one would compare the consensus times of multiple lattice sizes (with the shared initial condition $x = 0.5$) and long-range interaction probability p held fixed. Note that while x is held fixed at 0.5 in both types of measurement, any value of x can be selected without loss of generality or accuracy.

In this study, we will comment on the functional form of consensus time vs. p using the τ_2 data. Consensus (also sometimes called relaxation) time studies on the baseline SM have shown $\tau(x, \boldsymbol{p} = 0, L) \propto L^2$ [2], while studies on an SM with a "mass media effect influence probability" r indicate $\tau(x, p = 0, L) \approx L^{F(r)}$, where $F(r)$ indicates some function of r [12].

The Low-p, High-L limit: The Need for Parallel Computing

Another important aspect of our study is the wish to apply the highly productive and powerful techniques of parallel computing to simulate the lattices, thereby allowing multiple processors to work on a system simultaneously. The primary motivation for

parallel computing is the very slow simulation times characteristic of the $p < 0.1$ and $L > 50{,}000$ regime. Using certain parallel programming approaches, parallelizing the simulation code is beneficial across the entire spectrum of p-values.

In our simulations we employ two forms of parallelism, the first of which is formally called domain decomposition (DD). An explanation of DD is as follows. Whereas in a serial program one processor monitors and updates a lattice of L sites, in a DD approach the program farms out the lattice-updating responsibilities to P processors (or processor cores, we use the terms interchangeably here). Ultimately, each processor is tasked with doing the pair selections and m(t) calculations for its "sub-lattice" of size L/P. Although we do not do this, some more sophisticated algorithms incorporate load-balancing, allowing for sub-lattice domains to shift in order to unburden busier processors and make more use of the less busy ones [13, 14]. Moreover, our DD code is written exclusively for the 1D case; there is no possibility of a long-range site being selected for update.

Speaking generally, this parallelization technique should make the simulation process faster as there are many processors attacking a give lattice size, and each one is handling a lattice smaller than the original one. It is important, however, not to neglect the inter-processor communication and synchronization that must occur when sub-lattice boundary sites are selected. Additionally, considering the opposite direction (using fewer and fewer processors), we note that there is less synchronization to be done, but each sub-lattice being updated is significantly larger. What we just discussed is an overview of the general DD procedure, and we will comment on the critically important synchronization methods employed in constructing the program in the **Methods** chapter.

The second parallelization method that we employ is officially known as functional decomposition or task parallelism, although it is often referred to as "poor man's parallelism" (PMP) due to its simplicity to implement. Although less sophisticated than the DD method, it offers much more flexibility. Rather than splitting the work by dividing up the lattice (of size L) under study, we distribute the work by trial iteration. In effect, given N iterations at a particular x-value with P processors working on the simulation, each processor will do the following work during its N/P iterations at each x-value:

- pair selections,
- $m(t)$ calculations,
- file-writing operations (if necessary),

all for the whole L-sized lattice. The N boundaries present in each lattice (and thus in each trial/iteration) in the DD approach are gone in PMP, so there is much less synchronization that must occur. In fact, our PMP program incorporates no synchronization at all, and we will explain its exact operation in **Methods**.

To gain a quantitative understanding of the power of the parallel programs, it is necessary to perform benchmarking tests to determine how much of a speedup benefit they provide, if any at all. To be more specific, we note that in a simulation run by a parallel program, the parameter set we introduced before— just (L, p) with $x = 0.5$—now becomes (P, L, p). With these three parameters in mind, one now has a simple variable framework in which to benchmark the parallelization approaches. For example, P can be varied, while L and p are kept fixed, and we call this approach the following:

> **DPS/S (Different P Speedup/Slowdown)**: For the DD approach, as the ratio of the number of boundary sites to the number of interior sites decreases, the inter-

processor communication time should decrease and thus the ratio of inter-processor communication time to actual lattice updating time should decrease as well, leading to a simulation speedup. Nevertheless, DD is a dual-edged sword, as a smaller amount of boundaries does mean that each sub-lattice to be updated is larger (and more time-consuming to evolve). PMP is much simpler; it is a matter of raw horsepower. By splitting the *x*-iteration trials amongst more and more processors, we ask "how much less time does it take to run an entire simulation?"

Chapter 2 - Methods

An Overview of the Serial Code

To begin this section, we note for the reader that copies of the extensively commented serial code, DD approach code, and PMP approach code can be found in Appendices A-C, all written in C99. To obtain a detailed understanding of how the simulations were made to incorporate the dynamical rules of the LRSM, it is advantageous to first look at the full serial code in Appendix A.

Nevertheless, in the interest of greater clarity, a shortened, pseudo-code version of the serial code be seen below in Figure 2-1. Both of the parallel codes are built off this serial platform, so after explaining this basic structure, we will explain the specifics of each approach by noting the differences each has relative to this basis.

Figure 2-1. The pseudo-code version of SM1.c. Note that all variables here (with the exception of the tracking variables) follow the same naming convention as used in the introduction. *number_up* and *number_down* correspond to the number of up-opinion and down-opinion sites, respectively. The lattices are represented in memory as one-dimensional integer arrays, where each index can have value 0 (down opinion) or 1 (up opinion).

```
// include header files
// variable initializations and lattice array creation (lattice pointer named lat)
// random number seeding
// file pointer initializations, etc.
// L, p, and N established by this point in the program
for (x = 1; x < 100; x++) {
    U = 0;
    for(j = 1; j < N + 1; j++) {
        x_val = (double)(x) / (double)100;
        number_up = 0, number_down = 0;
        randomize_lattice(lat, &number_up, &number_down, x_val);
```

```
            evolve_lattice(lat, &number_up, &number_down);
            // evolves lattice from ICs to consensus 1 time (1 trial). m(t) tracked in here, too
            if(number_up == L) {
                U = U + 1;      // equivalent to U += 1;
            }
}
Exit_prob = (double)U / (double)N;
fprintf(Exit_prob_file, "%f %f\n", i, Exit_prob);
}
// file closing operations and program cleanup
```

The function names in this short code segment are meant to illustrate their functions' purpose. In SM1.c (aside from the initializations and program cleanup) we loop through the x-values, where at each x-value we run N trials of the lattice size and p-value under study. In each iteration, *randomize_lattice*() makes sure the intial conditions are satisfied and that the up-opinion sites are distributed randomly.

To make the random site selections, which establish the site pairs as well as the long-range sites to be updated, we use the built-in C function *rand*() (this is random enough, but a function from the Mersenne Twister algorithm/library would be even more truly random) [15]. For the randomly picked sites that are located within the first or last three indices of the lattice, we make sure to pick the appropriate sites on "the other end" of the lattice to obey the periodic boundary conditions. From there, *evolve_lattice*() drives the system from initial conditions to consensus one time, and then once the N trials are complete for a particular x, we insert U into Eq. (1) to compute the exit probability $E(x)$, which is then printed to a file (*Exit_prob_file*).

An Overview of the Parallel Codes

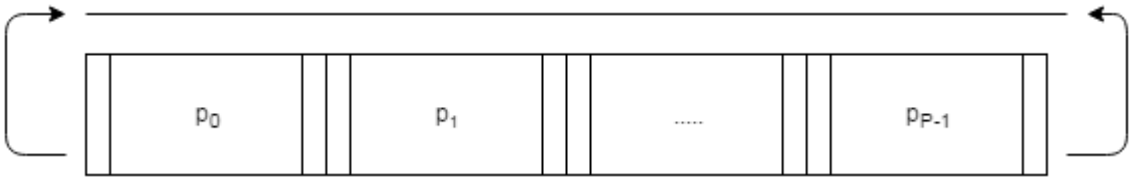
To make the parallel programs portable enough to work on any supercomputing architecture, we employed the C-language Open MPI (Message Passing Interface) version 1.8.0 parallelization library in both codes. While more complicated than its

shared-memory colleague/competitor OpenMP (which assumes a computing platform where every processor has access to the same data), Open MPI (O-MPI) is more portable due to its ability to work in shared memory and distributed memory environments (where each processor operates on its own data, and must communicate with the others to "have access to" their data) .

To give some more context on the operation of an Open MPI program, we will note that when an Open MPI program is launched (most simply with a command line statement like *$ mpiexec -np P ./C_PROGRAM*), the *P* processors called all receive the exact same executable *C_PROGRAM*. Then, depending on what the program is supposed to do, the programmer must write the proper logic in the code so that the process topology (how the processors are overlaid on top of the object on which they are operating, such as the lattice in MPI_1.c) does not result in "site possession" conflicts and that synchronization permits actions (site updates) to occur in the desired order.

In the case of MPI_1.c, our process topology (domain decomposition) takes the form seen in Figure 2-2.

Figure 2-2. The process topology of MPI_1.c, for P processors being called. Each processor operates on L/P sites (a sub-lattice). The narrow bands on the sides of each sub-lattice are boundary areas in which any updating must involve communication between the two "adjacent" processors. The arrows on the ends indicate that this is a periodic lattice (a chain).



Over the lifetime of an entire simulation, from $x = 0.01$ to $x = 0.99$, (0 and 1 need not be included since their exit probabilities must be 0 and 1, respectively), there are $99 \times N$ trials if x is incremented by 0.01. Within each x-value, we specify the j^{th} trial uniquely

by the ordered pair (*x,j*). The nested *for*-loop structure which MPI_1.c inherits from SM1.c can be seen in Figure 2-3 below.

Figure 2-3. The Pseudocode of MPI_1.c (*MPI_Barrier*()'s argument is the group of working processors) An individual trial begins with the *for*-loop on line 4, and ends after line 13. *assign_init_ups*() ensures that if the number of initial up sites does not distribute evenly across the processors, the "leftovers" will be randomly assigned. Here, *evolve_lattice*() does one pair selection rather than a whole trial.

```
// include header files
// variable initializations and lattice array creation (lattice pointer named lat)
// MPI initializations
// random number seeding
// file pointer initializations, etc.
// L, p, and N established by this point in the program


measure_begin_time();  // grab the starting time of the simulation
1 for (x = 1; x < 100; x++) {
2     U = 0;
3     assign_init_ups(x);
4     for(j = 1; j < N + 1; j++) {
5           MPI_Barrier(COMMUNICATOR_GROUP); // synchronize the "stepping" of the processors through the code
6           x_val = (double)(x) / (double)100;
7           number_up = 0, number_down = 0;
8           randomize_lattice(lat, &number_up, &number_down, x_val);
9
10             while(totalnumber_up != 0 && totalnumber_up != L/P) {
11               evolve_lattice(lat, &number_up, &number_down); // 1 inv. MCS
12               MPI_AllReduce(&number_up, &totalnumber_up,....); // find total number of ups among all sub-lattices
13          }
14
15          if(number_up == L) {
16               U = U + 1;
17          }
18    }
19    Exit_prob = (double)U / (double)N;
20    fprintf(Exit_prob_file, "%f %f\n", i, Exit_prob);
}
measure_total_time();   // use starting time and current time to determine total simulation time
// file closing operations and program cleanup
```

When each processor, running at its own speed, is calling e*volve_lattice*(), it does so with no knowledge that its sub-lattice is part of a larger lattice. Since we want the processors to be tackling the same physical (*x,j*) lattice simultaneously, we must force a synchronization at each (*x,j*) pair. This was done by putting *MPI_Barrier*() at the beginning of the second *for*-loop. This works because *MPI_Barrier*() statements have an effect such that all of the processors in COMMUNICATOR_GROUP will hold off on

executing the barrier-following statements (that is, the sub-lattice evolution commands for a particular (x,j) lattice) until all processors have reached the barrier.

We now have a motivation and a methodology for the time synchronicity of MPI_1.c, but we still have two facets of the program which greatly differentiate it from SM1.c. The first is the change in the operation of *evolve_lattice*(). Rather than doing one trial, it does one single pair selection within a sub-lattice. In this way, when a processor is going through the *while*() loop, it selects one site pair, updates the states of sites and opinion counts (if UP1 is satisfied), and then pools its up-opinion count (*number_up*) with the *number_up* values of the other processors using the *MPI_Allreduce*() command. The sum of all *number_up*s is inserted into *totalnumber_up* for each processor; if it is equal to zero or L, all the processors know that the collective lattice has reached consensus, and so they move onto the next (x,j) lattice.

The second (and last) major point of difference for MPI_1.c can be inferred from the process topology shown in Figure 2-2 and the distributed memory architecture of the Open MPI programming model; that difference comes from the selection of site pairs near the boundaries of the sub-lattices. When such sites are picked, there must be synchronizing communication between the neighboring processes. We illustrate the need for such an action with an example (it is important to keep in mind here that this example is not the only border-site selection that would give cause for synchronization).

Suppose processors J and K are neighbors in the process topology, where J is the processor on the left, and K is the processor on the right. Each processor picks a site pair by first selecting one site at random, site RSN, within its sub-lattice. RSN will be the leftmost site of the (A,B) site pair which will eventually do the updating. Then, if

processor J picks the rightmost site (index MAX) in its sub-lattice as RSN, the leftmost site of of Processor K's sub-lattice (index ZERO) must be the other half of the pair, site B. To determine whether the pair will follow the update rules of UP1 or UP2, it is necessary to fix the opinion of site B during the check; that is, its memory location must be locked during the check. To do this, there must be some sort of mutually acknowledged and abided-by form of locking between the two processors.

Before going into the specifics of that locking, we must give a general overview of the passive communication protocol we employed in our programs. To elaborate, Open MPI allows for processors to gain access to the information in other processors' memories via window objects overlaid on specified memory ranges. These memory ranges are allocated using *MPI_Alloc_Mem*(), given a size by being instantiated with data structure of a certain size (such as an integer array), and then are formally made into windows with the *MPI_Win_Create*() function. Since the programmer does not have to write in active call-and-respond commands for each process to "read the information" from the others or "write to their memory" (not reading and writing in the strictest sense of the words) this is called passive remote memory access (PRMA).

Now for the locking. Open MPI has associated locking and unlocking mechanisms for these windows (*MPI_Win_lock*() and *MPI_Win_unlock*(), respectively) which can envelop critical sections (code segments only allowed to be executed by one process at a time due to the nature of the data being updated in that segment), guaranteeing that a processor's constituent data members are only being updated by one processor at a particular moment. *MPI_Win_lock*() and *MPI_Win_unlock*(), while highly useful, do come with extra time overhead due to the fact that they will not return until a

lock has been successfully achieved; if a window is a prime area for being updated, all of the different processors will have to wait their turn to acquire their lock and do their updating or reading operations. It is important to note here that only Open MPI windows have this locking feature. Memory locations not designated as window objects do not enjoy this "safety" benefit.

In MPI_1.c we created our integer arrays (the sub-lattices) in each of the processes, but only overlaid windows on the three rightmost and leftmost indices (sites). The motivation for this selective windowing can be illustrated by considering the differences in update requirements for the sites illustrated by Figure 2-4 and those not covered by Figure 2-4. As a general statement, we note that the fundamental "mission" of the locking is to ensure that the status of all four sites involved (A,B,C, and D) remain stable (i.e., controllable) during a pair selection and update.

Figure 2-4. Zooming in on the boundary sites between adjacent processors in the process topology. RSN for Processor X is given by RSNX. Since in a pair selection, RSN/A is always on the left, one can see from the figure that these done-by-another-processor updates are possible: MAX can be updated by K when RSNK = MIN, MIN can be updated by J when RSNJ = MAX - 1, and MIN + 1 can be updated by J when RSNJ = MAX.

Let us consider the six possible ways that pairs can be selected from the sites in Figure 2-4 (six, one way for each of the possible values RSN could be). Remember here that RSN becomes A once the other half of the pair (that is, B) has been selected, so we will refer to it as RSN or RSN/A. It is important to note here that we are not assuming that RSN/A and B satisfy UP1 (they agree) at this point. With that, let us then go through the different cases.

If processor J set RSN = MAX – 2 (far left index), then B would be MAX – 1, MAX - 3 would be C (C will always be our leftmost site of the panel), and MAX would be D (D will always be our rightmost site of the panel). If A and B satisfied UP1, then we would need to lock D's memory location to update it—otherwise it might end up with the opinion of MIN and MIN + 1 instead. Similarly, if RSN/A was set to be MAX - 1, then MAX would be B (accessible by K). In this case, we would first need to lock MAX/B's memory location in order to verify if UP1 was satisfied; if so, we would have to lock both processor's windows to update MAX - 2 and MIN. In another scenario, If RSN/A was set to be MAX, then MIN would be B. As with the previous case, we would have to lock both windows to see if A and B satisfied UP1 (since A and B are *very* susceptible to being changed), and if they did, continue those two locks to update MAX - 1 and MIN + 1.

If RSN/A was chosen to be MIN, MIN + 1 would be B, MAX would be C, and MIN + 2 would be D. As with the other cases, the status of all four of these sites, in their different windows, must remain stable during a pair selection and update, so we must lock K's window for the pair selection, and then both windows for the updates if UP1 is satisfied. Continuing with the Processor K sites, if RSN/A was set as MIN + 1, MIN + 2 would be B, MIN would be C, and MIN + 3 would be D. MIN and MIN + 1 are the only ones that can be updated by Processor J (see Figure 2-4), so we only need to lock K's window for this RSN/A selection and site update. Finally, if RSN/A = MIN + 2, MIN + 3 would be B, MIN + 1 would be C, and MIN + 4 would be D. MIN + 1 is the only index accessible for update by Processor J, so we only lock the Processor K window here, too.

There is a small note to be made here. Since we always update sites C and D if A and B satisfy UP1, we do the opinion counts for each processor after each call to *evolve_lattice*() by looping through the sub-lattice and counting the number of each opinion. The lattices' two windows are not locked during this procedure, so it is possible that the boundary sites can be updated during the counting. Ultimately, this is not a problem because we have the check in our code (in the *while*-loop) to make sure that every site in the whole *L*-lattice is in agreement before moving to the next trial.

In the previous paragraph we showed that as we move RSN/A away from the border between Processor J and Processor K, that locking at least becomes less complicated, if not important. For RSN/A < MAX - 2 and RSN/A > MIN + 2 there is no possibility of a shared ability to update a given site, and so there is no need to have windows (and use their locking features) for such sites. Normal assignment operations and checking will do. Having windows over these inner sites is certainly still valid, though.

With the operation of MPI_1.c covered, we now move on to our second parallel program, MPI_2.c (incorporating PMP). While it lacks the finesse of MPI_1.c, it is much more streamlined, lightweight and it allows us to get more exit probability precision because of the greater number of trials possible at each *x*-value. The pseudo-code version of it can be seen in Figure 2-5 below.

Figure 2-5. The pseudo-code version of MPI_2.c. It has a structure very similar to MPI_1.c.

```
// include header files
// variable initializations and lattice array creation (lattice pointer named lat)
// MPI initializations
// random number seeding
// file pointer initializations, etc.
// L, p, and N established by this point in the program

measure_begin_time();
```

```
1 assign_iterations(N);       // burden processor 0 with the extra iterations (usually a negligible amount)
2 for (x = 0; x < 100; x++) {
3     U = 0;
4     for(j = 0; j < N; j++) {
5           x = (double)(i) / (double)100;
6           number_up = 0, number_down = 0;
7           randomize_lattice(lat, &number_up, &number_down);
8           evolve_lattice(lat, &number_up, &number_down); // does one trial, m(t) tracked in here
9
10          if(number_up == L) {
11                U = U + 1;
12          }
13    }
14    fprintf(Exit_prob_file, "%d %d %d %d\n", processor_id, U, x, N);
}
measure_end_time();
compute_sim_time(end_time, begin_time);   // find the total simulation time
// file closing operations and program cleanup
```

At a first glance, this code looks identical to SM1.c, with the exceptions of the statements *assign_iterations*(N) and *fprintf*(Exit_prob_file, "%d %d %d %d\n", processor_id, U, x, N). The former function is used divide the number of iterations among the processors equally if N mod P = 0, or, if N mod P ≠ 0, to divide as equally as possible among the processors and then give the leftover iterations to processor 0. We did not use more than 20 processors for a given (L,p) parameter set in this study, and N was never less than 500; then, noting that the maximum value of N mod P is P -1 and that P << N, we see that processor 0 does not get too much extra work in N mod P ≠ 0 scenarios.

The latter command is a slightly refined version of the identically located *fprintf* statement in SM1.c. To appreciate the significance of this new print statement, though, we have to first explain the significance of the PMP approach. With the task parallelism employed in this code, our objective is to have each of the processors do (approximately) N' ≡ N/P iterations at each x-value. Fortunately, unlike DD, there is no need for the processors to be in lock-step working on the same (x,j) lattice since each processor moves independently through the range of x-values. At the end of its N' iterations (trials) at a

given x-value, a processor prints how many of the iterations ended as unanimously up. With 3 processors running, each having N' = 100, the output to the exit probability file could look something like this:

1 14 5 100
0 15 5 100
0 14 4 100
1 12 3 100
2 17 3 100
2 13 4 100

An explanation of the data formatting is as follows. The first column consists of identifiers for each processor (identifier values ranging from 0 to $P - 1$), the second column is the particular x-value the processor just ran the N' trials for, the third column is the number of times the lattice evolved to be unanimously up (U from Eq. (1)), and N' makes up the fourth column. Although the fourth column may seem redundant in this scenario, in other cases where N' is not the same for all processors, this is important (see Eq. (1)).

To find $E(x)$, we simply sum $U(x)$ for each of the processors (i.e., $U_1(x) + U_2(x) + \ldots + U_p(x)$), and then divide that sum by $N_1' + N_2' + \ldots + N_p' = N$, where N_k' is N' for the k-th processor. All 99 x N trials are independent of each other, so we can wait until all of the raw U data comes in from all of the processors to evaluate $E(x)$ from 0 to 1. It is also important to note here that some of the rows are out of order; this is not an inconvenience, but rather a benefit. This shows that some processors are moving faster than others, not being throttled by the slower ones because the speed bump *MPI_Barrier*() commands present in MPI_1.c are not present in MPI_2.c

These three programs allow us to find the raw $E(x)$, program run time, and consensus time data (contained in the m(t) data). From this data, we can easily find the speedup of the parallel programs as a function of the number of applied processors P.

Finding Critical Exponents from the Simulation Data

To analyze the raw exit probability and consensus time data, we also designed and wrote a program to numerically determine the critical exponents that would best fit the simulational data. Written in MATLAB, we labeled it the Critical Exponent Finder (CEF), and a copy of it can be found in the Appendix D. In this section we will describe its operation, first via a pseudocode description, and then with a supplementary explanation.

Figure 2-6. The CEF algorithm, CEF.m.

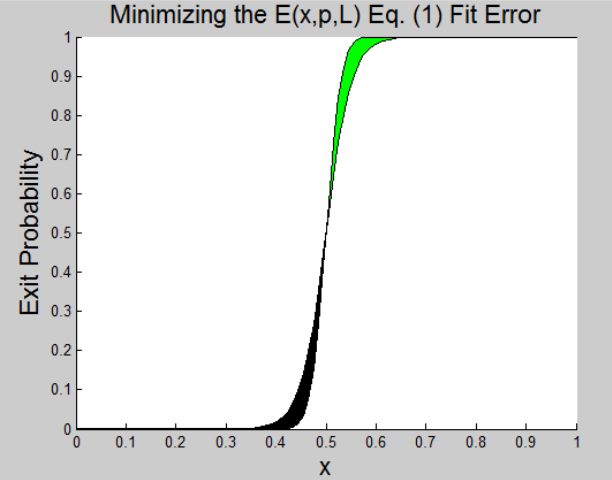
```
LAMBDAs = [0:0.01:5]                                % create array of N_lambda lambda values
NUs = [0:0.01:5]                                    % create array of N_nu nu values
L_VALS = [100, 500, 1000, 5000, ... etc.]           % create array of system sizes
ERR_MATRIX = zeros(N_lambda, N_nu)                  % store the cum. errors across system sizes for each fit
error_max = 10000                                   % start out with a huge possible fit error

% All L-value E(x) data from simulations done at a particular p (data files have been imported at this point)
for (j in LAMBDAs)
        for (k in Nus)
          % lambda/nu pair has been picked at this point
          % now we try out lambda/nu pair for each of the L-data sets
          lambda_nu_error_sum = 0;   % cum. error across sytem sizes for l,v pair
        for (i in L_VALS)
                size_error_sum = 0;
                for (x in X_VALS)
                   L = L_VALs(i)
                   LAM = LAMBDAs(j)
                   NU = NUs(k)
                   EP_EXPT = EXIT_PROB(L_VALs(i),x)                    % grab raw E(x) data
                   EP_THEOR = ½(tanh(L^(1/NU)*LAM*2*(x - 0.5)) + ½     % compute E(x) using exponents
                  size_error_sum += (EP_EXPT - EP_THEOR)^(2)           % sum error across x-values, fixed L
                end                                                    % x_values loop finishes
        lambda_nu_error_sum += size_error_sum;
        end                                                            % end of L-values loop
if (lambda_nu_error_sum < error_max)
    error_max = lambda_nu_error_sum;
    best_lambda = lambda_value;                                        % recalculate best lambda-nu pair
    best_nu = nu_value;
end
ERR_MATRIX(j,k) = lambda_nu_error_sum;    % insert exponent pair cum. error (across system sizes) into matrix
end                                       %end of nu loop
end                                       % end of lambda loop

[r_min, c_min] = find(ERR_MATRIX==min(ERR_MATRIX(:)))          % find location of minimum-error pair
fprintf("Best lambda is %f and Best nu is %f\n", LAMBDAs(r_min), NUs(c_min))
meshgrid(ERR_MATRIX^(-1))                                      % display the heatmap (surface)
```

At its core, CEF is a least squares fitting algorithm, where we create $N_{lambda}*N_{nu} = N_{pairs}$ versions of Eq. (3). Each of those N_{pairs} equations is laid on top of the $E(x)$ data for a given system size, and the area of the shaded sections in Figure 2-7 is computed.

Figure 2-7. A Pictorial Representation of the operating guidelines of the CEF. The smallest area between the two curves indicates the greatest agreement between the curves possible.



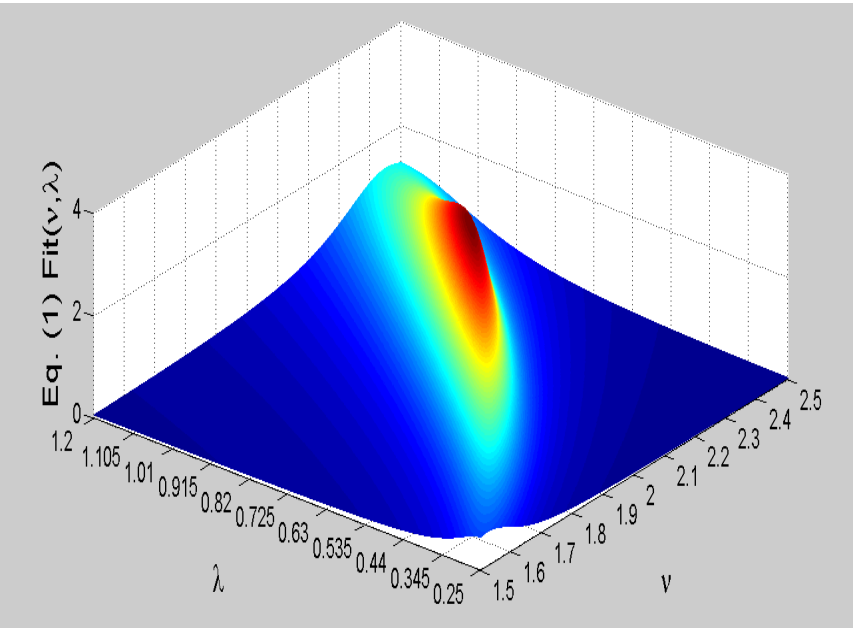
Those areas are summed up across the different system sizes at a given p, and the λ,ν pair that results in the smallest area is selected as the critical exponent pair for that p.

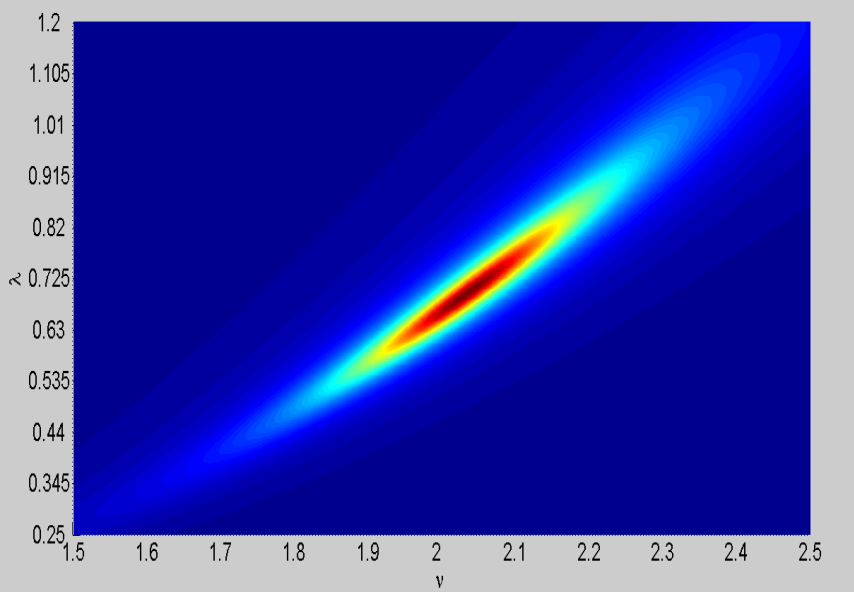
To give a basic overview of the operation of the program, the first two *for*-loops select a critical exponent pair. Then, in the third (running through L-values) *for*-loop, the area between the two curves (experimental data and theoretical) is established across the x-values for a given system size. The cumulative error of each system size's data with Eq. (3) is summed, and then once all system sizes have been scanned through that final critical exponent error is stored in an error matrix (*ERR_MATRIX*). The most universally acceptable λ,ν for a particular p is then found by just selecting the minimum value of *ERR_MATRIX*. While the least squares algorithm that we use to find the critical

exponents as a function of *p* is a perfectly acceptable one, others have also been developed for this purpose [16].

One way that we obtain further insight into the critical exponents via the CEF program is using heat surfaces (heatmaps in two dimensions) to give a three-dimensional (3D) representation of what *ERR_MATRIX* looks like. A heatmap is a pictorial representation of a two-dimensional matrix, where each matrix element is assigned a color (out of a discrete but many-membered color distribution) based on the value of the datum contained there; frequently, red (hot) corresponds to larger values, while blue (cool) corresponds to smaller values. By inverting each element of *ERR_MATRIX*, we allow the low-error critical exponent pairs to be easily visible red spots on the heatmap, while high-error pairs are a dark blue color. An example of a heatmap is shown in Figure 2-8.

Figure 2-8. An example of a heatmap (p = 0.5, λ = 0.701, ν = 2.037). The above image is a 3D view of the heat "mountain," while the lower figure is a purely top-down view if it.





The heatmaps alone are not sufficient to determine critical exponent pairs, but they are an excellent way to ensure that there is only one local maximum in the range of λ and ν specified in CEF.m. If there were multiple red peaks, that would indicate multiple ways to describe the $E(x)$ phase transition at a given p, which would be problematic.

In addition to the numeric approach of using a least squares algorithm to find the critical exponents that would best agree with simulation data, we performed series expansions of Eq. (3) (the tanh( ) function) and Eq. (4) (the sigmoidal polynomial) to see what limiting values of the critical exponents, if any, would be required for the two functions to give the same result in the 1DL.

Chapter 3 – Results and Discussions

In this study, we investigated the limiting behavior of $\lambda(p)$ and $\nu(p)$ as $p \rightarrow 0$, the speedup in execution time for the simulations run under the DD (data level) and PMP (task level) parallelism schemes in the DPS/S testing framework, the consensus time τ_2

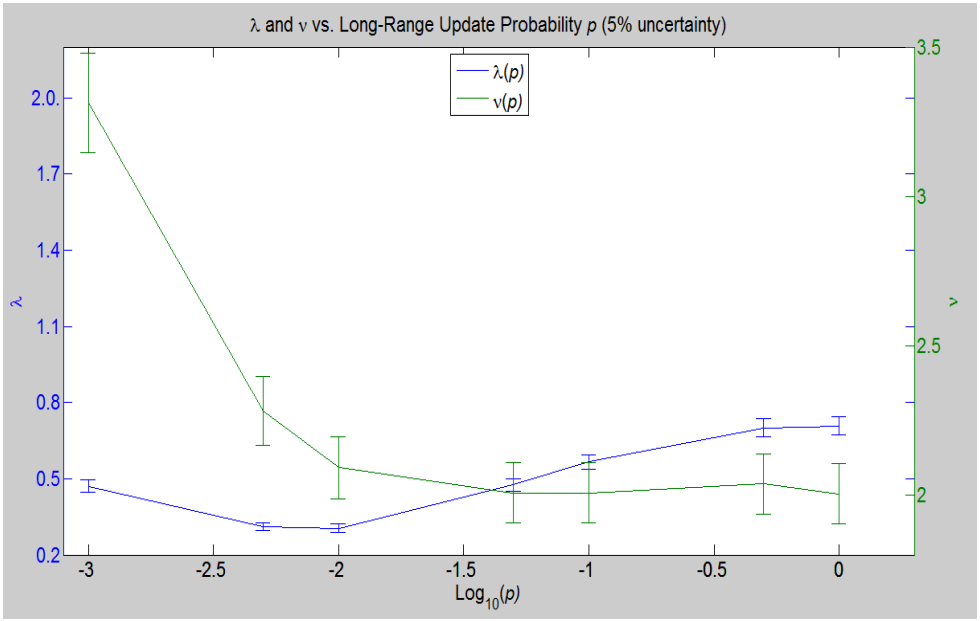
for a fixed system size, and the general dependence of system $m(t)$ (magnetization vs. time) vs. p for the LRSM.

We first discuss the primary focus of our study: the limiting behavior of the critical exponents λ and ν. To find $\lambda(p)$ and $\nu(p)$, we applied the CEF algorithm to exit probability data from a set of simulations that includes various (p,L) combinations (that is, each L-value sized lattice is subjected to 14 different program implementations, one for each p-value). The available p and L values are listed below:

- L = {100, 500, 1000, 5000, 10000, 50000, 100000}
- p = {0.00, 0.001, 0.0025, 0.005, 0.0075, 0.01, 0.025, 0.05, 0.075, 0.1, 0.25, 0.50, 0.75, 1.00}

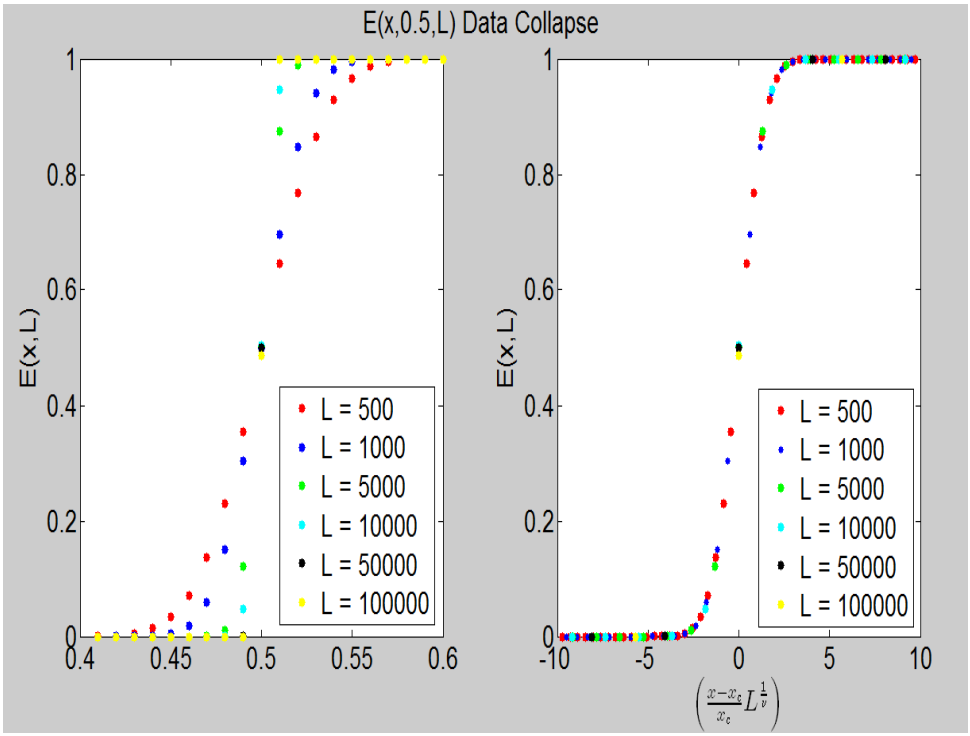
From the CEF analysis, we produced the plot shown in Figure 3-1 (next page), which details the p-dependence of λ and ν. When viewing the plot, it is important to remember that the critical exponent at each p-value is based on the results of an algorithm that sums errors (between the tanh(λ ,ν) function and the simulation data) across the exit probability data for seven different system sizes. For the higher-p simulations this cumulative error is not so large due to the smoothness of the data curves allowed by the high-N nature of the simulations (10000 or greater). The low-p exponents ($p \leq 0.005$) have lower iteration counts ($O(2000)$) at each x, so the data there is more "jittery," meaning a larger overall error when CEF is comparing the data to the smooth tanh(λ ,ν) curves.

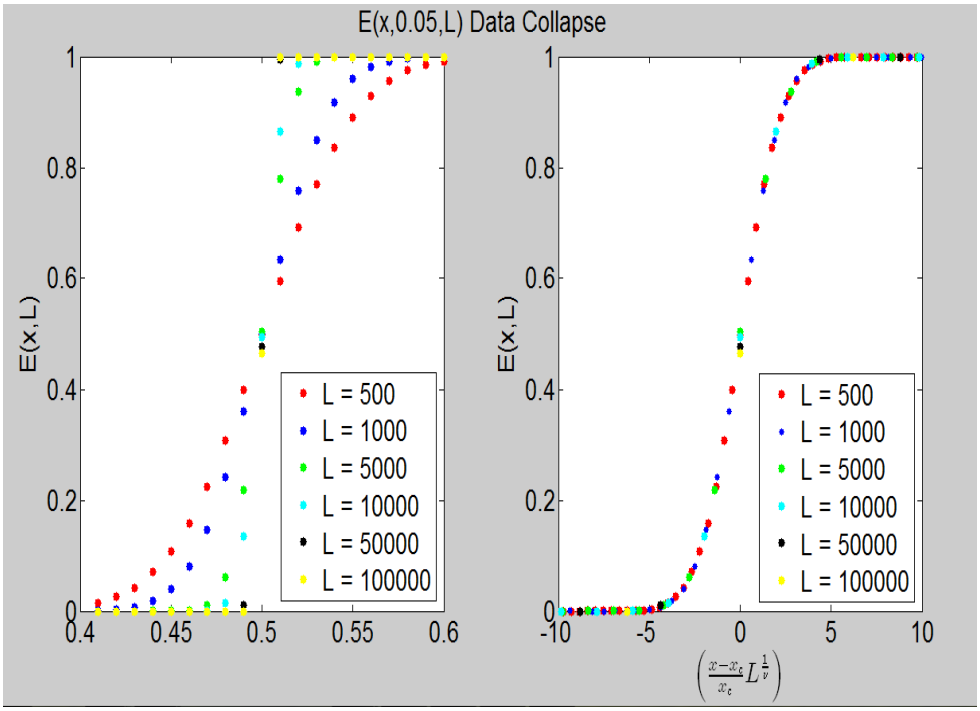
Figure 3-1. $\lambda(p)$ and $\nu(p)$ as $p \to 0$. (note the base-10 abscissa axis and linear ordinate axis; when plotted with linear ordinate and abscissa axes, the divergent features are more pronounced) While λ stays relatively constant as p approaches 0, there is a strong diverging trend for ν. The vertical bars represent the 5% uncertainty in the exponent values. Additionally, not all p are present, their simulations having not yet completed. The lines are only to guide the eye.



Neither of the exponents is constant across p, but the two do change in a continuous fashion, indicating that the overall nature of the phase transition is constant in p. The non-monoticity of their behavior is likely due to insufficient number of runs—this will be investigated further with more runs and more p-values. The excellent data collapse illustrated in Figure 3-2 (next page) demonstrates that these critical exponents work well in describing the discontinuous phase transition observed for all $p>0$.

Figure 3-2. Data collapse of exit probability for L ≥ 500 at p = {0.5, 0.05}. In the above figure, $p = 0.5$ has the critical exponents λ = 0.701, ν = 2.037, and below, $p = 0.05$ with λ = 0.475, ν = 2.007. In both cases the incredibly strong collapse indicates that the pairs are highly suitable for describing their respective $E(x)$ phase transitions. $L = 100$ is not included for the sake of graph clarity.



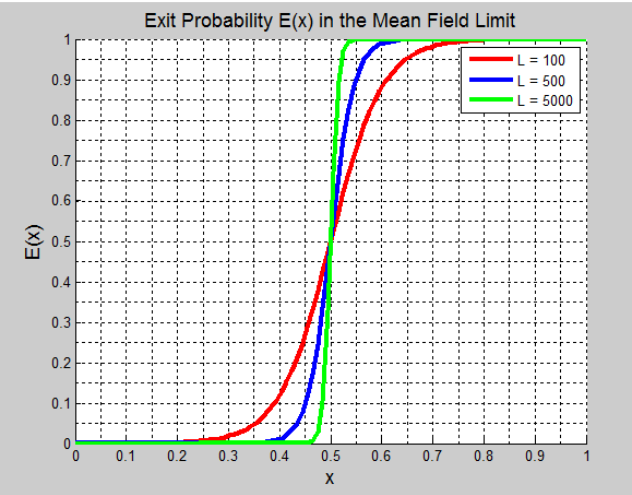


Based on the $p>0$ critical exponents fitting a step function exit probability, the next natural questions is: as these exponents tend toward their limiting behavior (∞ and approximately 0.5 for ν and λ, respectively), how does the $E(x,p,L)$ curve for the LRSM change, and is that functional form correct for the $p = 0$ limit?

To (at least partially) answer that, we refer back to the limiting cases analysis of Eq. (3) which we framed in **An Explanation of The Topics of Interest.** There we noted that "as $\lambda \rightarrow 0$, $E(x,L)$ becomes a strongly linear and increasingly flattening function still passing through the point (0.5,0.5)" and "as $\upsilon \rightarrow \infty$, $E(x,L)$ becomes increasingly linear, still passing through (0.5,0.5)." Since λ stays approximately constant, while υ is sharply diverging, we can conclude that overall, $E(x,p,L)$ should be tending toward a roughly linear form near $x = 0.5$.

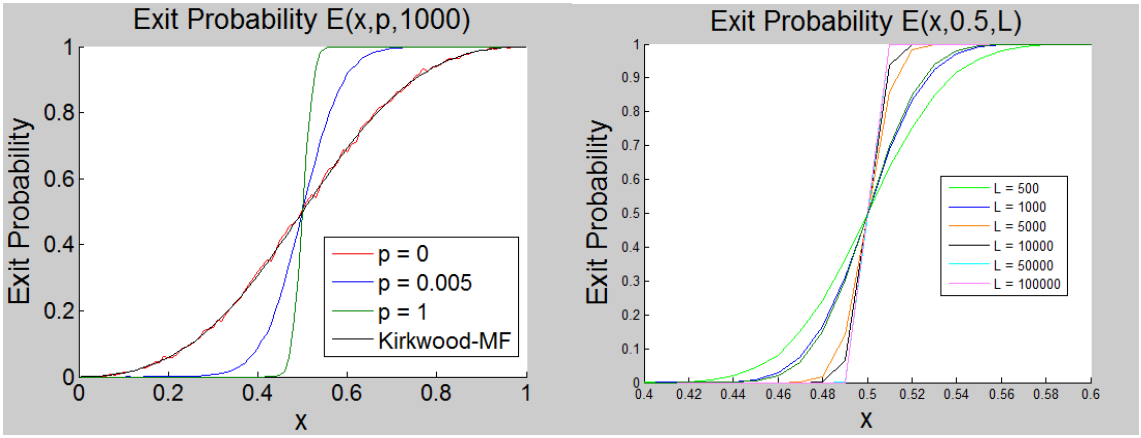
This gives a good general picture of what $E(x,p,L)$ looks like in the 1DL, but it is not a fully descriptive one by any means. To get a more complete picture of the LRSM functional form, we momentarily consider approaching the MFL. In that regime, $\nu \rightarrow 2$ and $\lambda \rightarrow 0.7$; for $L = 100$, 500, and 5000, we reach the three $E(x,L)$ curves plotted below.

Figure 3-3. E(x,p,L) for the MFL limit ($\nu \cong 2$ and $\lambda \cong 0.7$). $L = 100$ is the red line, $L = 500$ is the blue line, and $L = 5000$ is the green line. Note the tendency toward a step function in the thermodynamic limit.



From Figure 3-3 we see that the exit probability in the MFL displays a strongly steplike behavior, especially for L-values on the order of 10^4 and above. Then, putting the analysis of the 1DL and MFL exit probability behavior together we can reasonably predict that $E(x,p,L)$ should be a roughly s-shaped curve that moves from step function to an s-curve shape as we move from the MFL to the 1DL. In Figure 3-4 below, which contains our simulation results, one can see that prediction to be consistent with the actual data.

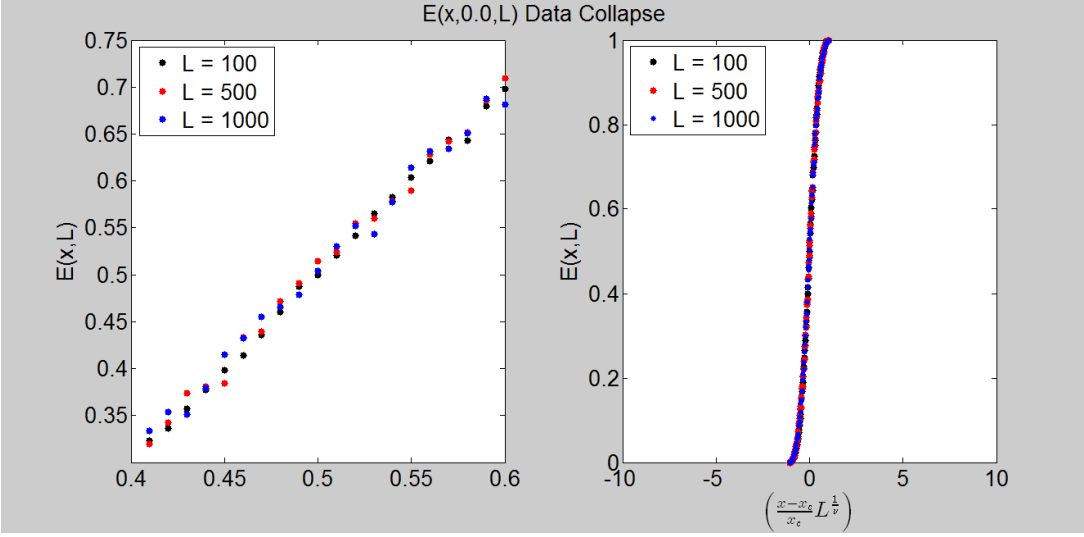
Figure 3-4. The Step-function behavior of Exit Probability in the Mean-Field limit. The left-hand figure shows the tendency of $E(x,p,L)$ to move towards step-function behavior as the 1DL is approached. Indeed, although high-p simulations move a particular L-value exit probability curve to the step-function domain, the right-hand figure shows that increasing system size has the same effect. In effect, a large system size (or high p-value) moves a system to the "majority-rules" consensus behavior, while a smaller system size (or lower p) allows for somewhat of a "success-proportional-to-presence" consensus behavior.



We have shown that $E(x,p,L)$ for the LRSM moves toward an s-shaped curve in the 1DL. Then, looking at the left-hand side plot of Figure 3-4, we see that the $p = 0$ exit probability data for $L = 1000$ (other system sizes, too; we only show $L = 1000$ here for the sake of graph cleanliness) is essentially on top of the Kirkwood-Mean Field approximation; that is, Eq. (4) shows a strong, statistically valid agreement with the original SM, independent of system size as predicted.

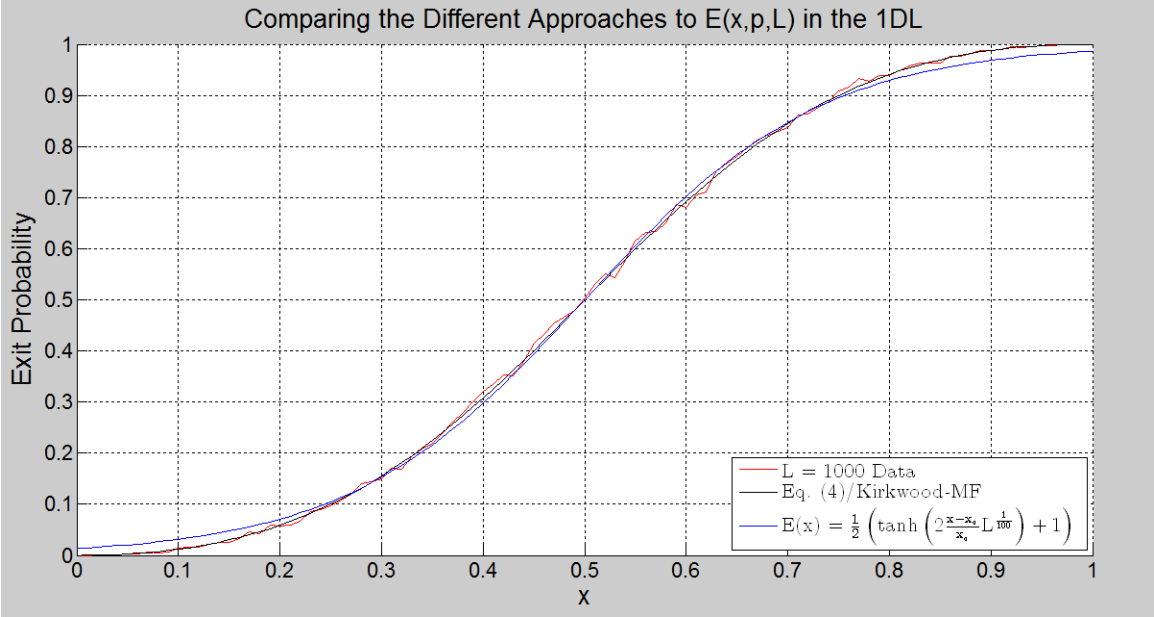
From the previous figure and our simulation data it is plain to see that Eq. (4) is valid for the LRSM in the 1DL. Then, if Eq. (3) is to be consistent with the data, it must be equal to Eq. (4) when the limiting exponents ν(0) and λ(0) are inserted into it. Although we indicated approximately what ν and λ were trending toward before, we will more officially note now that λ(0) □ 2.1 and that 100 is a suitably accurate value of ν(0). With these exponents inserted into Eq. (3), we found the strong data collapse seen in Figure 3-5.

Figure 3-5. Data Collapse for the 1-Dimensional Limit. On the left-hand plot pane, E(*x*,*p*,*L*) is plotted from x = [0.4, 0.6] to emphasize the tight correlation amongst the curves associated with the different lattice sizes. All are very close to passing through (0.5,0.5), indicating that these are high-precision, satisfactory runs. On the right-hand pane we show the actual data collapse; note the strong overlap, indicating that the phase transition is well described the exponents. The rescaled *x*-interval is smaller due to the large magnitude of *ν*.



Plotting the tanh( ) function with these exponents and overlaying the resulting curve on top of the ubiquitous form of Eq. (4) and the L = 1000 simulation data we reached the following plot in Figure 3-6 (next page). As with the left-hand pane plot of Figure 3-4, we only included one lattice size for clarity.

Figure 3-6. A Comparison of the mean-field and ansatz tanh( ) exit probabilities in the 1DL. That is, comparing Eq. (3) and (4) to the L = 1000 simulation data with the 1DL exponents plugged into Eq. (3).



From Figure 3-6 we can see that all three graphs are almost on top of each other in the range $0.25 \leq x \leq 0.75$, but only Eq. (4) is consistently in agreement with the L = 1000 data, indicating that it is overall a better fit to the 1DL. Ultimately, when parameter fitting the tanh( ) equation, there is a trade-off between satisfying the boundary conditions, which push $E(x)$ to be more step-like (underestimating exit probability for $x < 0.5$ and overestimating it for $x > 0.5$) and fitting the intermediate x-values, which make the function more linear (but push $E(x)$ away from satisfying the boundary conditions).

In our alternate approach to finding the critical exponents in the 1DL, we performed Taylor series expansions of Eqs. (3) and (4) about $x = x_c$. We chose to expand about x_c due to its central location within the range of x which allowed us to create a maximally covering functional form (over the the initial up-opinion spectrum) with the least amount of series terms. Additionally, Eq. (3), with its tanh( ) argument having the format (constants)×(x - x_c) is naturally set up for an expansion about x_c. Once the

expansions were made, we created a set of equations, equating each $(x - x_c)^k$ coefficient (A_k) for the two equations. The expansions can be seen below.

First Eq. (3):

$$E(x,L) \cong \frac{1}{2} + \lambda L^{\frac{1}{\nu}}\left(x - \frac{1}{2}\right) + \frac{4}{3}\left(\lambda L^{\frac{1}{\nu}}\right)^3\left(x - \frac{1}{2}\right)^3 + \frac{32}{15}\left(\lambda L^{\frac{1}{\nu}}\right)^5\left(x - \frac{1}{2}\right)^5 + \sigma(x^7)\# \quad (5)$$

And then Eq. (4):

$$E(x) = \frac{1}{2} + 2\left(x - \frac{1}{2}\right) - 8\left(x - \frac{1}{2}\right)^3 + 32\left(x - \frac{1}{2}\right)^5 + \sigma(x^7) \quad (6)$$

In a first order approximation (x^1), we can see that $\lambda L^{1/\nu} = 2$, however, if we go to higher order corrections in the series— $O(x^9)$, for example— $\lambda L^{1/\nu}$ approaches a value of approximately 1.65. Since each A_k term is dependent on λ, ν, and L, it is impossible to find a "panacea" pair of critical exponents that fits every single system size in the context of a particular p. Earlier in this section we noted that at every single p, there was a cumulative error across the L-spectrum when fitting the tanh( ) function to the simulational data; this is due to the noise in the simulational data which comes with a non-infinite N and the data's x-value granularity ($\Delta x = 0.01$), of course, but on a more fundamental level, it is a consequence of the tri-variable dependence of the tanh( ) function's expansion coefficients.

Taking one last look at Eq. (6), we must mention the following: if one supposes that $\lambda L^{1/\nu} = 1.65$, and we assume λ is constant and small (on the order of 1), it follows that the bigger the system size, the larger ν must be to satisfy the relation. This is supportive, though not necessarily definitive proof for, a diverging value of ν.

Moving away from the critical exponents, we now turn to the internal workings of the "black-box" LRSM, and consider the behavior of the second commonly measured consensus time dependence, $\tau_2(\boldsymbol{x}=0.5, p, \boldsymbol{L})$, where the bolded-italic notation signifies that the quantity is held fixed. However, before directly finding a functional form to describe τ_2 vs. p, it is natural and illuminating to first study the dependence on p that the magnetization $m(t)$ (for a particular system size) has. With this in mind, we conducted simulations with $L = 1000$ and $p = 0.005$, 0.5, and 1.0, finding the following $m(t)$ behavior (Figure 3-7).

Figure 3-7. m(t) for $L = 1000$, $x = 0.5$, and $p = 0.005$, 0.5, 1.0. Each data line is one trial. $m(0) = 0$ since we have an equal number of up and down sites at that time. Note that the final magnetization of the lattice determines whether U is incremented by one or not. For example, if we had done 500 iterations with $L = 1000$, $x = 0.5$, and $p = 0.005$, there would be 500 curves on the plot. If 246 of them ended with $m(t) = 1$, E(0.5,0.005,1000) would be 246/500 = 0.492.

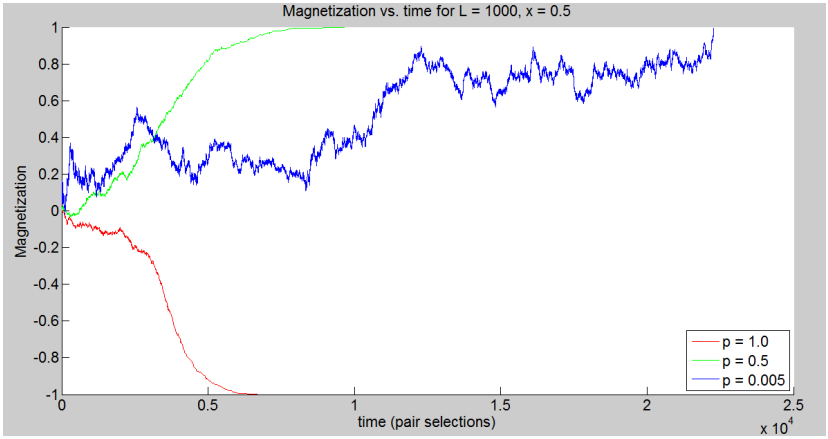
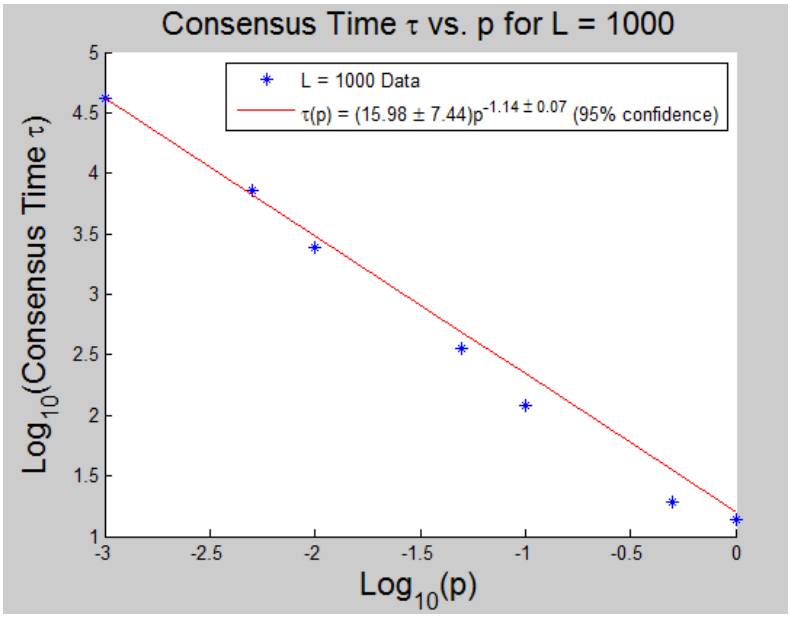


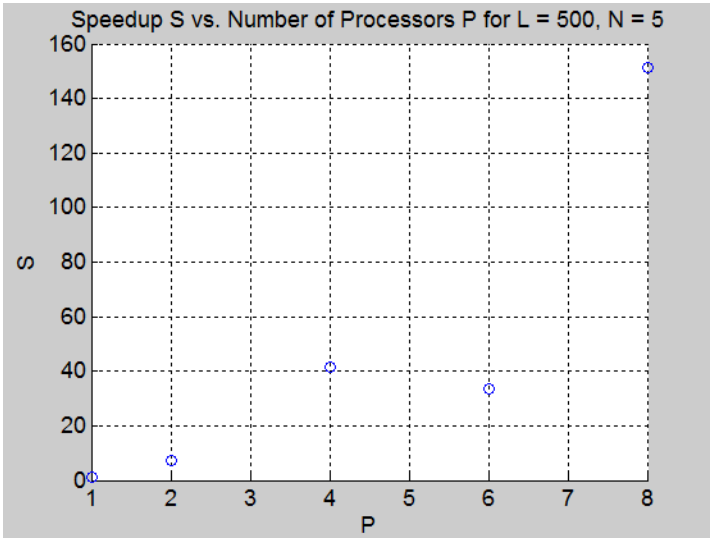
Figure 3-7 provides a good qualitative sense of how p affects consensus time; looking at the abscissa, we can see that (in terms of pair selections) for $p = 1$, $\tau \approx 70000$ (70 MCS), for $p = 0.5$, $\tau \approx 100000$ (100 MCS), and for $p = 0.005$, $\tau \approx 220000$ (220 MCS). Then, expanding this to include all of the other p-values except $p = 0$ (due to the failure of the base-10 logarithm there), we reached the following figure (Figure 3-8, next page).

Figure 3-8. The consensus time $\tau_2(x=0.5, p, L=1000)$. $\tau(p) \propto p^{-1}$ for $p > 0$. Note that both axes are base-10 logarithmic.



At this point we now move onto our results from the speedup tests, and we will first discuss our results from the DD approach. A plot of the speedup results for this parallelization approach, with $L = 500$, $p = 0$, $N = 5$, and $x = 0.25$ can be seen below.

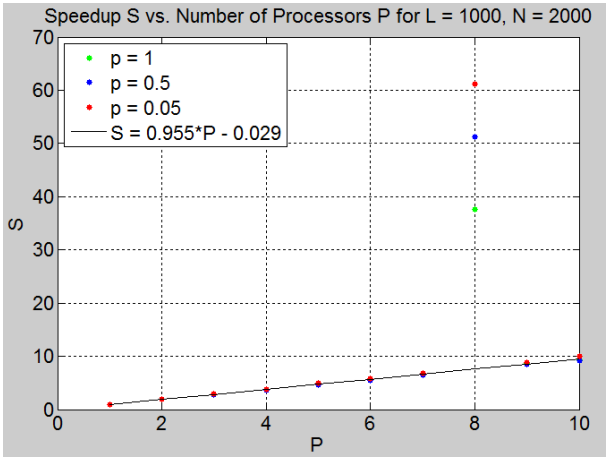
Figure 3-9. Program Speedup S vs. P processors (data level parallelism) Here, $L = 500$, $p = 0$, $N = 5$, and $x = 0.25$. Note that we only have used even numbers of processors (except $P = 1$). This is to avoid the slight irregularity of sub-lattice sizes that comes with odd-P values. For the actual speedup measurement, we ran small test runs of five iterations at the chosen x, and then repeated this small five-bundle of trials 10 times. We then averaged the $10*P$ data points (where a data point is the time it took a processor to run the five-bundle) to find the average time it took a given process to run five trials.



Right away we can tentatively suppose a superlinear (greater than P) speedup as we increase P. Without the presence of the odd-P runs, it is hard to get an exact functional form of the peedup, but it would seem reasonable to consider a possible exponential speedup vs. P. The slight downturn at $P = 6$ is possibly an artifact of the low number of runs which we used. More specifically, we conducted these small 5-bundle runs with such a small number of iterations so that we could keep a close eye on the simulations as they were running. To be more realistic and account for the variances in simulation times that are intrinsic to any stochastic simulation, it is best to have larger N for the iteration counts (at least a few hundred). Finally, including odd P-values as well would help complete the functional form of speedup with the DD approach.

The other parallelization benchmark test that we performed was with the PMP code. Running simulations with the parameter set $L = 1000$, N = 2000, and just focusing on the iterations at $x = 0.5$, we found the following results for the speedup vs. the number of applied processors.

Figure 3-10. Program Speedup S vs. P processors (task level parallelism). Each data point is based on the average time that it took a given processor in the simulating group to finish its iterations at $x = 0.5$. In turn, the coefficients of the linear fit S(P) are the average of the coefficients from the linear fitting of each of the data series (for $p = \{1, 0.5, 0.05\}$, and $P = 8$ data excluded).



When considering the linear regression S(P), the 0.955-value coefficient for P makes sense. With $N = 2000$ and the low values of P, processor 0 was burdened with a negligible number of extra iterations. Since each process was essentially doing the same number of iterations, and there were no synchronization barriers, the time that it would normally take one processor to conduct the simulations was simply split among the P processors, leaving a speedup of P. Some reasons that can explain why the slope is less than the expected 1 include:

- The fact that this was conducted on a multi-processing system (running CentOS 5.7) where processors can occasionally be interrupted by other tasks, thus preventing them from "working" 100% of the time
- It is possible on some of the odd-P simulations that the few extra runs processor 0 received modified the overall simulation time average.
- Some processors are inherently faster than others, and this skews the average processor simulation time sometimes.
    - Expanding on this idea, only evaluating Speedup at $x = 0.5$ may not have left us enough statistical room to account for the inherent variance in processor speeds.
- The time-keeping operations we used in MPI_2.c might not be precise enough, or there may have been a loss of accuracy somewhere in the program's execution.

There are many other possibilities for what might have caused this small deviation from the expected result, but given how close the slope coefficient is to 1, we are pleased. In a final note, we could have forced the ordinate-intercept value to be 0 (ours was a consequence of linear fitting in MATLAB's cftool), changing the slope coefficient, too.

One highly abnormal feature to come out of the DPS/S testing for the PMP approach was the consistently greater speedup at $P = 8$, which seemed to only grow with decreasing p. After doing much debugging and repeatedly testing to see if this was a fluke, and not seeing the peak go away, we were forced to come up with some possible answers for its origin.

The most probable cause of this behavior is at least partially based on the optimizations that the ACG has set in place for its cluster of 32 nodes, where each node contains 16 physical cores. Specifically, for jobs where 8 processes are requested, the job server (PBS) finds one particular node where 8 cores are available, and sends the job there. Even in high-performance computing, geography is still important, and the closer processing units are to each other, the better (even in this scenario of minimal inter-process communication, where all processors must deliver output to the same unit).

One reason for such an optimization might be that users commonly specify 8 processes for MPI programs, and so the administrators wanted those to be fast—or, it could be that each node contains two 8-core processors. In the latter case, that is even more motivation for this choice; now, processes do not have to necessarily relay information from vastly different nodes and cores to the file location where the timing data is stored. There is one "pathway" from the processes to the storage place, leading to a tight correlation between processor execution times.

Chapter 4 - Conclusions

In this study we introduced long-range interactions to the Sznajd Model via a probability p analogous to the dynamic and static small world rewiring parameter. Using parallel and serial Monte Carlo simulations, we employed finite size scaling analyses in

characterizing the exit probability for $p \neq 0$, finding step function behavior that was well described by a tanh( ) function relying on two p-dependent exponents. Indeed, as p approached zero, the exponents changed (in a continuous fashion) indicating a constancy in the nature of the (discontinuous) phase transition for $p > 0$. Ultimately, for all $p > 0$, we found excellent data collapses for the critical exponents found.

By analyzing the one-dimensional limit of the exponents in the ansatz tanh( ) function, and comparing the tanh function evaluated with those exponents to the sigmoidal polynomial which has already been shown to be successful in the 1DL, we sought to determine if the ansatz function was valid in the one-dimensional realm.

We found that these exponents, although one (ν) was divergent in the 1DL (and much larger than the value determined by B & S), fit the $p = 0$ simulation data well and provide an excellent approximation for $0.25 \leq x \leq 0.75$; nevertheless, the sigmoidal polynomial is still a better fit with the simulational data. The reason for the good agreement between the tanh( ) prediction and the simulational data in the middle of the x domain but not near the ends is due to the trade-offs of trying to optimally fit λ and ν.

Indeed, although we have tried many p-values in our studies of the LRSM, there are some low-p runs (0.0075, 0.0025, 0.075, 0.025, for example) which still need to be finished. Upon evaluating the data from these low-p runs and doing some higher N iterations of our current low-p datasets (using the parallel programs), we hope to have a more complete picture of these exponents so that we can more definitively say what their 1DL behavior is. Based on the current graphical behavior of $\lambda(p)$, we suspect that it will stay on the order of 1 as more data comes in. Until then, the presence of one strongly diverging exponent (not a physically intuitive scenario) leads us to the conclusion that

there is a fundamental break at $p = 0$; the exit probability phase transition about $x = x_c$ changes at that p and the tanh( ) function is not truly valid there. Finally, we also find that consensus time at fixed system size is proportional to $p^{-1.1}$.

In testing the parallel code that helped in the data gathering for the $p = 0$ runs, we found that a task level parallelism approach (PMP) worked well, with a speedup very near to being directly proportional to the number of processors applied to the simulation; an exception to this general trend occurred at $P = 8$, where the cluster's batch system optimizations localized the processes to one node, allowing for a much greater speedup.

The data level parallelism approach (DD), although only applied at five P-values (1,2,4,6,8), did show signs of superlinear speedup. However, the lack of odd-P values and the low Ns used in this approach limits our ability to set a concrete functional form for the speedup data. One other thing to note is that the DD approach code was written exclusively for the $p = 0$ case. Any p-value greater than that would require each sublattice to employ locks even when interior sites are being updated. With so many more locks being now employed, the programs would slow down dramatically. For this reason, we believe that the PMP approach is a more practical one to take—it is adaptable to any p, it is easy to implement, there is no processor synchronization required, and the speedup is very easily assured.

We have seen that the Sznajd Model in its various implementations (whether on the stock trading floor, in Brazilian elections, including long-range interactions, or even in a non-binary political party system) has achieved success in predicting the outcomes decided by large numbers of individuals. In conducting this study where we have

attempted to shed more light on the 1D question of whether the exit probability is truly sigmoidal (derived from mean-field considerations which should not work) or if it is a step function in the thermodynamical limit (and the sigmoidal shape seen is due to finite size effects), we conclude that the sigmoidal polynomial is indeed correct for $p = 0$. We have employed traditional and parallel simulation methods, finite scaling analysis to judge the fit of the exponents found at each p, and have investigated the effect of p on consensus time (good for ballpark simulation time estimates). In doing this we hope that we have laid a groundwork for future consideration of long-range two or three-dimensional Sznajd models.

Bibliography

[1] K. Sznajd-Weron and J. Sznajd, Int. J. Mod. Phys. C 11, 1157 (2000).

[2] F. Slanina, K. Sznajd-Weron, and P. Przybyła, Europhys. Lett. 82, 18006 (2008).

[3] K. Sznajd-Weron, Acta Physica Polonica B 36, 2537 (2005).

[4] K. Sznajd-Weron and J. Sznajd, Practical Fruits of Econophysics 355 (2006).

[5] D. Stauffer, Advs. Complex Syst. 05, 97 (2002).

[6] K. Sznajd-Weron and R. Weron, Int. J. Mod. Phys. C 13, 115 (2002).

[7] D. E. Rodriguez, M. A. Bab, and E. V. Albano, Phys. Rev. E 83, (2011).

[8] P. Roy, S. Biswas, and P. Sen, J. Phys. A: Math. Theor. 47, 495001 (2014).

[9] P. Przybyła, K. Sznajd-Weron, and M. Tabiszewski, Phys. Rev. E 84, (2011).

[10] T.E. Stone and S.R. McKay, Physica A: Stat Mech and Its Apps 419, 437 (2015).

[11] T.E. Stone and S.R. McKay, EPL (Europhysics Letters) 95, 38003 (2011).

[12] N. Crokidakis, Physica A: Statistical Mechanics And Its Applications 391, 1729 (2012).

[13] L. Kalé, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten, Journal Of Computational Physics 151, 283 (1999).

[14] B. Hess, C. Kutzner, D. V. D. Spoel, and E. Lindahl, J. Chem. Theory Comput. 4, 435 (2008).

[15] Random Number Generators - The PLab Project – Literature. <http://random.mat.sbg.ac.at/literature/>

[16] S. M. Bhattacharjee(l) and F. Seno, J. Phys. A: Math. Gen. 34, 6375 (2001).

Appendix A: The Long-Range Sznajd Model (LRSM) code (SM1.c)

SM1.c was written in C99, compiled with gcc and run in a Linux environment (Ubuntu 14.04 LTS). The instructions on how to compile and run the program with the desired parameters are listed right below the #*include* commands. *time*(NULL) (which returns the number of seconds passed since January 1st, 1970) is used to seed the random number generator due to the fact that it allows for different random number selection each time the program is run. The comments which aim to help clarify the code will generally either be to the right or above the statements they are explaining (for all Appendices). Also, note that the parameters (L, p) are hard-coded into the code, so each time the parameters are changed, the program must be recompiled for the new values to take effect.

```
#include <stdio.h>
#include <math.h>
#include <time.h>        /*include the necessary header files*/
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

// To compile: gcc –o SM1 SM1.c  (SM1 is the executable)
// To run: ./SM1 (parameters are hard-coded in)

//CAVEAT FOR THIS CODE. MAKE SURE THAT L and DELTA X ARE OF INVERSE
ORDERS OF MAGNITUDE OR LESS. THAT IS, IF THERE ARE 10^3 SITES, DELTA X
//CANNOT BE SMALLER THAN 10^-3, OTHERWISE THE "C" VALUE WILL ALWAYS
BE TRUNCATED TO ZERO AND THE CODE WILL HAVE THE APPEARANCE OF AN
INFINITE
//LOOP. JOSEPH GARCIA, 9/29/2014

typedef struct       /*creating the struct which will symbolize an individual site*/
{
        int state;            /*state == 1, site is up, state == 0, site is down*/
} site;

int Num_of_sites = 1000;  /* number of sites in the system, also known as the lattice size */
```

```
double p = 0.01;   /* probability of updating a long range neighbor */
int num_of_runs = 20000;       /* number of trials per x-value. The more, the better */
int mcs_runs_to_select = 20;    /* number of trials to select for Monte Carlo Step timing data */

int random_num_interval(const int Num_of_sites);     /*random number generation function*/

/* zeroes the array during each trial at a particular x-value (makes it all down sites) */
void zero_array(site func_array[Num_of_sites], int counter);

/* used to order the 20 randomly selected trials */
int compare_ints (const void *a, const void *b);

int main()            /*begin the main function*/
{

int mcs_start_index = 0;          /* keeps track of trials at which MCS data is recorded */
int record_magnetization = 0;  /* flag indicating whether MCS data should be recorded or not */
double EofX = 0.00;                        /* E(x) */
int c=0,k=0,f=0,g=0,v=0;  /* c=number_up initially, k = delta x indexing value,
                                  f,g,v are tracking variables in reserve */
double y = 0;        /* variable for displaying MCS information*/
double x = 0;                  /* x */
int number_up = 0;          /* number of up sites at any point in a trial  */
int number_down = 0;          /* number of down sites at any point in a trial */
double magnetization = 0;      /* magnetization at a particular time */
int selected_site = 0;        /* randomly chosen site to make up during the init/randomization */
int rand_site_num = 0;        /* randomly selected site during the updating process (site A) */
int start_site2 = 0;          /* neighboring site of rand_site_num (site B)*/
int left_neighbor = 0;        /* left neighbor of rand_site_num (C) */
int right_neighbor = 0;       /* site to the right of B (D)*/
int number_up_sum = 0;        /* the U in Eq. (1) */
const int u = 1;              /* constant signifying site is oriented up */
const int d = 0;              /* constant signifying site is oriented down */
int MCS_count = 0;            /* number of Monte Carlo steps for a particular trial */
double deltaX = 0.01;                /* goes from 0 ->1, the smaller deltaX is the more
                                        data points we have, leading to a smoother curve. */
const int Num_of_slots = 1.000/deltaX;   /*total number of x-values to scan through */

int random_time = time(NULL);    /* obtain time in seconds since the UNIX Epoch */
srand(random_time);            /* seeding the pseudorandom number generator */

FILE * output = 0;              /* output file pointer for all E(x,L) information */
FILE * output_MCS = 0;          /* output file pointer for all MCS information */
FILE * output_MAG = 0;          /* output file pointer for all magnetization information */

char Ns[10];
```

```
char ps[10];                    /* hold LRSM parameters as strings for file name purposes */
char Ns[10];
char total[1000];
char * preceder = "SM1_Data/N";
char * c_time_string;           /*  variable declarations for several other strings in the file
names */
time_t current_time;
char month_day[50];

current_time = time(NULL); /* Obtain current time as seconds elapsed since UNIX Epoch */
if (current_time == ((time_t)-1)) {
  (void) fprintf(stderr, "Failure to compute the current time.\n");
  return EXIT_FAILURE;
}
c_time_string = ctime(&current_time); /* Convert to local time format. */
if (c_time_string == NULL) {
  (void) fprintf(stderr, "Failure to convert the current time.\n");
  return EXIT_FAILURE;
}

memcpy(month_day, &c_time_string[4], 6);   /* establish month/day section of file names */
memset (&month_day[3],'_',1);   /* insert an underscore between the month and day words*/

sprintf(Ns, "%d", Num_of_sites);
sprintf(ps, "%f", p);                   /* convert parameters into character strings */
sprintf(Ns, "%d", num_of_runs);


/* snprintf establishes file name for exit probability data file */
snprintf(total, sizeof(total),"%s=%s_Runs/%s_%s_%s_SM1_NC_%s.dat", preceder, Ns, Ns, ps,
Ns, month_day);

/* open the file for writing (appending to current data or opening the file if it doesn't exist */
output = fopen(total, "a");

/* snprintf establishes file name for MCS data file */
snprintf(total, sizeof(total),"%s=%s_Runs/%s_%s_%s_SM1_MCS_%s.dat", preceder, Ns, Ns,
ps, Ns, month_day);

/* open the file for writing (appending to current data or opening the file if it doesn't exist */
output_MCS = fopen(total, "a");

/* snprintf establishes file name for magnetization (vs. t) data file */
snprintf(total, sizeof(total),"%s=%s_Runs/%s_%s_%s_SM1_MAG_%s.dat", preceder, Ns, Ns,
ps, Ns, month_day);
```

```
/* open the file for writing (appending to current data or opening the file if it doesn't exist */
output_MAG = fopen(total, "a");

clock_t start;          /*create clock variable to capture time the simulation begins*/
clock_t finish;         /*create clock variable to capture time the simulation ends*/

site sitestruct[Num_of_sites];   /*array representing the one-dimensional lattice of sites*/
/*int MCS_info[num_of_runs];*/       /*array containing information regarding the MCS counts*/

/* array of 20 indices corresponding to trials to be selected for MCS information gathering */
int mcs_rand_selections[mcs_runs_to_select];  /

start = clock();    /*capture the simulation beginning time*/

for (k =0; k < (Num_of_slots + 1); k++)  /*BEGIN FOR_LOOP_1 (the x-values loop) */
{
     x = deltaX*k;          /* the x-value */
     number_up_sum = 0;         /* no trials have resulted in a homogenously up system yet. */
     MCS_count = 0;             /* trials have not yet started */

     if((x !=0) && (x !=1))  /* no simulations necessary for x = 0 or x = 1 */
     {
     for(f = 0; f < mcs_runs_to_select; f++) {
          mcs_rand_selections[f] = 0 + (rand() % (int)((num_of_runs - 1) - 0 + 1));
         /* elect random iterations for MCS data gathering */
     }

/* put the random trials in increasing order */
     qsort(mcs_rand_selections, mcs_runs_to_select, sizeof(int), compare_ints);

/* initialization of tracker determining how many iterations have undergone MCS data logging */
     mcs_start_index = 0;

     for(f = 0; f < num_of_runs; f++) /*BEGIN FOR_LOOP_2 (the iterations loop for each x) */
     {
          zero_array(sitestruct,g);      /* zero the array of sites */
/* number_up, number_down are reset to safe zero-values */
          number_up = 0, number_down = 0;

           c = x*Num_of_sites;    /* number of sites to be initially up */
          MCS_count = 0;             /* rezeroing MCS_count, just to be safe */
          while(c>0)      /* BEGIN WHILE_LOOP_1 (initialize the lattice for each trial) */
          {
               selected_site = random_num_interval(Num_of_sites);  /* select a random site to set
                                                                      as up within the array */
                 if (sitestruct[selected_site].state == d)   /* make sure that site is "down" */
```

```
                {
                    sitestruct[selected_site].state = u;   /* make the down site up */
                    c = c - 1;                      /* decrement c, as one more site is now up */
                    number_up = number_up + 1;      /* increase by 1 the amount of up sites */
                    number_down = Num_of_sites - number_up;
                }
            }           /*END WHILE_LOOP_1*/
        if (f == mcs_rand_selections[mcs_start_index]) {
            mcs_start_index += 1;
            record_magnetization = 1;                /* if we are at an MCS-recording trial, set the
proper flags */
        }
        while((number_up < Num_of_sites) && (number_down < Num_of_sites))   /*BEGIN
                                                    WHILE_LOOP_2 (the consensus while loop) */
        {
         /*pick a random site in the array/ lattice */
          rand_site_num = Num_of_sites * ((double) rand () / ((double) RAND_MAX + 1));
            MCS_count += 1;   /* pair will be selected now, so increment step count by 1/L MCS */
           /* note: (ln = left neighbor, rn = right neighbor,  ss2 = adjacent site) */
          if(rand_site_num == Num_of_sites - 1)
                    /*Bookend condition 1:  [ss2 rn x x x x x x x x x ln rsn] */
                {
                    start_site2 = 0;
                    left_neighbor = rand_site_num - 1;
                    right_neighbor = 1;
                }
            if(rand_site_num == Num_of_sites - 2)
                    /*Bookend Condition 2: [rn x x x x x x x x x ln rsn ss2]*/
                {
                    start_site2 = Num_of_sites - 1;
                    left_neighbor = rand_site_num - 1;
                    right_neighbor = 0;
                }
            if(rand_site_num == 0)
                    /*Bookend Condition 3: [rsn ss2 rn x x x x x x x x x ln]*/
                {
                    start_site2 = 1;
                    left_neighbor = Num_of_sites - 1;
                    right_neighbor = 2;
                }
            else {
                    /*Standard Bookend Condition: [x x x ln rsn ss2 rn x x x x x]*/
                start_site2 = rand_site_num + 1;
                left_neighbor = rand_site_num - 1;
                right_neighbor = rand_site_num + 2;
             }
```

```
if(sitestruct[rand_site_num].state == sitestruct[start_site2].state)  /* If UP1 is satisfied */
{

    /* LR site for rsn  to update */
   int random_long = Num_of_sites * ((double) rand() / ((double) RAND_MAX + 1));

    /* LR site for ss2 to update */
   int random_long2 = Num_of_sites * ((double) rand() / ((double) RAND_MAX + 1));

    /* probability of rsn updating an LR site */
   double rand_probability = ((double) rand() / ((double) RAND_MAX));

    /* probability of ss2 updating an LR site */
   double rand_probability2 = ((double) rand() / ((double) RAND_MAX));

      if(rand_probability <=p)  /* rsn will update an LR neighbor (random_long) */
              {
                    /* make sure LR site is not same state as rsn */
                   if(sitestruct[random_long].state != sitestruct[rand_site_num].state
                   {
                    /* make LR site same state as rsn */
                        sitestruct[random_long].state = sitestruct[rand_site_num].state;
                              if(sitestruct[rand_site_num].state == u)
                              {  /* incrementing opinion counts accordingly */
                                    number_up = number_up + 1;
                                    number_down = Num_of_sites - number_up;
                              }
                              if(sitestruct[rand_site_num].state == d)
                              {
                              number_down = number_down + 1;
                              number_up = Num_of_sites - number_down;
                              }
                   }
              }

              if(rand_probability > p) /* rsn will update ln */
              {
                    /* make sure ln is not same state as rsn */
                   if(sitestruct[left_neighbor].state != sitestruct[rand_site_num].state)
                   {
                              /* make ln same state as rsn */
                        sitestruct[left_neighbor].state = sitestruct[rand_site_num].state;
                        if(sitestruct[rand_site_num].state == u)
                              {
                              number_up = number_up + 1;
```

```
                number_down = Num_of_sites - number_up;
                 }
            if(sitestruct[rand_site_num].state == d)
                 {
                number_down = number_down + 1;
                number_up = Num_of_sites - number_down;
                 }
      }
}

if(rand_probability2 <=p)   /* ss2 will update LR site random_long2 */
{
       /* make sure random_long2 is not same state as ss2 */
      if(sitestruct[random_long2].state != sitestruct[start_site2].state)
      {
                /* make random_long2 same state as ss2 */
            sitestruct[random_long2].state = sitestruct[start_site2].state;
            if(sitestruct[start_site2].state == u)
            {
                number_up = number_up + 1;
                number_down = Num_of_sites - number_up;
            }
            if(sitestruct[start_site2].state == d)
            {
                number_down = number_down + 1;
                number_up = Num_of_sites - number_down;
            }
      }
}

if(rand_probability2 > p)  /* ss2 will update rn */
{
       /* make sure rn is not same state as ss2 */
      if(sitestruct[right_neighbor].state != sitestruct[start_site2].state)
      {
       /* make rn the same state as ss2 */
            sitestruct[right_neighbor].state = sitestruct[start_site2].state;
              if(sitestruct[start_site2].state == u)    {
                 number_up = number_up + 1;
                 number_down = Num_of_sites - number_up;
            }
       if(sitestruct[start_site2].state == d)      {
                number_down = number_down + 1;
                number_up = Num_of_sites - number_down;
        }
      }
```

```
                    }
            } /* UP1 condition satisfied statement completion */
            if ((record_magnetization == 1) && (MCS_count % Num_of_sites == 0)) {
/* record MCS data at right trials */
                magnetization = ((double)number_up -
(double)number_down)/((double)Num_of_sites);
                  /* write data to the proper file */
                fprintf(output_MAG, "%f %d %f\n", x, f, magnetization);
            }
      }        /* END WHILE_LOOP_2  (the consensus while loop)  */
      record_magnetization = 0;   /* reset magnetization recording flag */
            if(number_up == Num_of_sites) {
                  /* increment number_up_sum if system evolved homogeneously up in the trial */
                number_up_sum = number_up_sum + 1;
            }
           /* calculate exit probability */
            EofX = (((double) number_up_sum) / ((double) num_of_runs
                if(x == 0.5) {
                  /* record  consensus time for the x-value == 0.5 */
                    fprintf(output_MCS, "%d %d\n", f, MCS_count);
                }
            }   /*END FOR_LOOP_2, (the iterations loop for each x) */
      }   /* end of the x == 0 or x == 1 selection statement */

      if(x == 0) {
            EofX = 0.00;    /* x == 0, then the lattice is never homogeneously up */
      }
      if (x == 1) {
            EofX = 1.00;   /* x == 1, the lattice evolves to be homogeneously up for every trial */
      }

           /* put data (which is in the NC format) into the E(x) file */
      fprintf(output, "%3f %3f\n", x, EofX);
           /* display to the screen mcs, x-value, (x,L), L */
      printf("%3f  %3f  %d\n", x, EofX, Num_of_sites);

}       /*End FOR_LOOP_1 (the x-values loop)  */

finish = clock();     /* capture the time that the simulation ends */
fclose(output);          /* close the file which contains E(x,L) information */
fclose(output_MCS);       /* close the file which contains MCS information */
fclose(output_MAG);   /* close the file which contains m(t) information */

/* compute how long the simulation took in hours, minutes, seconds */
double sim_time = (((double) (finish - start)) / (CLOCKS_PER_SEC));
```

```
/* print the actual time of simulation to the screen */
printf("\nTime elapsed was: %3f s = %3f minutes\n", sim_time, sim_time / (60));

return 0; /*return 0, since main() is an integer returning function that has executed properly */
}

/*actual implementation of the random integer number generator function*/
int random_num_interval (const int Num_of_sites)
{
    int t = 0;
    t = Num_of_sites * (((double) rand()) / ((double) RAND_MAX + 1));
        /*note that rand()/(RAND_MAX + 1) is always less than 1.*/
    return t;
}

/*go thorugh the array of sites, make every site down by default*/
void zero_array(site func_array[Num_of_sites], int counter)
{
    counter = 0;
    for(counter =0; counter < Num_of_sites; counter++)
    {
        func_array[counter].state = 0;
    }
}

/* comparison function for ordering array of MCS-recording trial numbers */
int compare_ints (const void *a, const void *b)
{
const int *da = (const int *) a;
const int *db = (const int  *)  b;
return (*da  > *db) - (*da < *db);
}
```

Appendix B: The Domain Decomposition Approach LRSM Code (MPI_1.c)

MPI_1.c was written in C99, compiled with mpicc (a wrapper around gcc) and run in a Linux environment (CentOS 5.7). The instructions on how to compile and run the program with the desired parameters are listed right below the #*include* commands. One important thing to remember in reading this code is that memory regions which are to later become MPI windows must be allocated using the *MPI_Alloc_mem*() function. Finally, we note here that *time*(NULL)

(which returns the number of seconds passed since January 1st, 1970) is used to seed the random number generator due to the fact that it has a unique value (and therefore allowing for different random numbers to be selected) each time the program is run.

```
#include <mpi.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>
#include <sys/time.h>
#include <sys/resource.h>


/* Here is what the command line implementation should look like: */
/* mpiexec -n 2 ./mpi2 TOTALSIZE P_VALUE NUM_OF_RUNS */
/*                [0]    [1]      [2]      [3]        */

#define XSIZE          100
#define UP             1
#define DOWN           0

/* lattice initialized to have only down sites */
void init_array(int sublattice[],int sublatticesize);

/* lattice's up sites are randomly distributed */
void randomization_init(int sublattice[], int * c, int * number_up,
int * number_down, int sublatticesize);

/* pick a random site within the sub-lattice */
int random_num_interval (int sublatticesize);

/* pick pair, set up updating conditions (for interior or boundary locations), and update sites */
void convert_to_global_indices_and_update(int sublattice[], int array[], MPI_Win
sublattice_win, MPI_Win sublattice_btm, MPI_Win sublattice_top, int * localstep);

/* function called when sites are not near the sub-lattice boundaries */
void update_near_neighbor(int lattice[], int array[]);

/* determine process topology */
void neighbor_determination(int taskid, int * low_neighbor, int * high_neighbor, int numtasks);

/* function called when sites to be updated are near the sub-lattice boundaries */
```

```
void update_near_neighbor_BCs(int sublattice[], int array[], MPI_Win sublattice_win, MPI_Win
sublattice_btm, MPI_Win sublattice_top, int * localstep);

/* random number generator, used to pick random sites in lattices */
long random_at_most(long max);

/* count the number of up and down sites in a sub-lattice */
void updown_ct(int * up, int * down, int state, int sublatticesize);

/* establish the indices of the site pair and the immediate neighbors */
int global_assignments(int global_ss, int * global_ss2, int * global_ln, int * global_rn, int
latticesize);

/* create the files for data writing, give them the proper file names */
void create_and_open_files(FILE * output, FILE * output_MCS, FILE * output_m, char *
latticesize, char * p, char * N);

/* helper function used to establish files names */
void concatenate_strings(char * string, char * message, char * latticesize, char * p, char * N);

/* used to randomly select a processor when leftover down sites need to be distributed */
int return_rand_proc(int numtasks);

void main (int argc, char *argv[])
{
if (argc != 4 ) {  /* check that program is being properly used */
      printf("You forgot to include one or more arguments.\n");
}

struct rlimit rlim = { RLIM_INFINITY, RLIM_INFINITY };
if ( setrlimit(RLIMIT_STACK, &rlim) == -1 ) {                  /* increase stack size */
      perror("setrlimit error");
      exit(1);
}

int random_time = time(NULL);    / *seed random number generator with Epoch time */
srand(random_time);            /* continuation of seeding the pseudorandom number generator */

/* NOTE: unmarked variables serve the same purpose as their serial program counterparts */
int    numtasks;       /* number of processors working on the simulation */
int    sublatticesize;  /* number of sites in the sub-lattice */
int    latticesize;       /* the snumber of sites in the original lattice under study */
double p;
int    N;
int *   sublattice;
int *   local_c;  /* number of down sites in a particular sub-lattice */
```

```
int    taskid;     /* identifier unique to each process (just an integer) */
int    i;
int    j;
int    x;
int    c;                  /* several for-loop tracking variables */
int    t;
int    k;
int    one = 1;
int    proc;
int    number_up;
int    number_down;
int    low_neighbor;           /* low neighbor is the "left" process in the process topology */
int    high_neighbor;        /* high neighbor is the "right" process in the process topology */
int    number_up_sum;
int    localstep;   /* a single 1/L MCS, done by one processor */
int    colltd_numup;    /* number of up sites (combined from all processors) */
int    colltd_numdn;  /* number of down sites (combined from all processors) */
int    MCS_count;
int    rsn;
int    c1;
int    remainder = 0;
double  x_value;
double  m;
float   sum;
double  globetime;
FILE *  output = 0;             /*output file pointer for all E(x,L) information*/
FILE *  output_MCS = 0;         /*output file pointer for all MCS information*/
FILE *  output_m = 0;           /*output file pointer for all m(t) information */

/* various numbers that each process should know, such as neighbors, numup, numdown, etc. */
int    array[14] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0};

clock_t start = clock();  /* grab simulation start time (want to include MPI initializations) */

/***** MPI Initializations *****/
MPI_Init(&argc, &argv);              /* initialize the MPI context */

/* use MPI_COMM_WORLD to determine number of tasks (processors) */
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

/* find out processor identifier */
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);

/* incorporate some error handling protocols into the MPI environment */
MPI_Comm_set_errhandler(MPI_COMM_WORLD,MPI_ERRORS_RETURN);
```

```
latticesize = atoi(argv[1]);
sublatticesize = atoi(argv[1])/numtasks;
p = atof(argv[2]);
N = atoi(argv[3]);
lattice = (int *)malloc(numtasks*sublatticesize*sizeof(int));

MPI_Win sublattice_win;    /* creation of MPI window object variables */
MPI_Win sublattice_btm;
MPI_Win sublattice_top;
MPI_Win local_c_win;

/* allocate memory locations for sub-lattice and local up-amount variables */
MPI_Alloc_mem(sizeof(int)*sublatticesize, MPI_INFO_NULL, &sublattice);
MPI_Alloc_mem(sizeof(int)*1, MPI_INFO_NULL, &local_c);

/* creation of the MPI windows overlaid on the allocated memory ranges */
MPI_Win_create(sublattice, sublatticesize*sizeof(int), sizeof(int), MPI_INFO_NULL,
        MPI_COMM_WORLD, &sublattice_win);
MPI_Win_create(sublattice, 3*sizeof(int), sizeof(int),
        MPI_INFO_NULL,  MPI_COMM_WORLD, &sublattice_btm);
MPI_Win_create(&(sublattice[sublatticesize - 3]), 3*sizeof(int), sizeof(int), MPI_INFO_NULL,
        MPI_COMM_WORLD, &sublattice_top);
MPI_Win_create(local_c, 1*sizeof(int), sizeof(int), MPI_INFO_NULL, MPI_COMM_WORLD,
        &local_c_win);

/* create and open the data output files */
if (taskid == 0) {
      create_and_open_files(output, output_MCS, output_m, argv[1], argv[2], argv[3]);
}

/* determine the processor's place within the process topology */
neighbor_determination(taskid, &low_neighbor, &high_neighbor, numtasks);

array[0] = taskid;
array[1] = numtasks;
array[2] = sublatticesize;
array[3] = 0;   /* number_up */
array[4] = 0;    /* number_down */
array[5] = 0;    /* ln (C) */
array[6] = 0;    /* ss/rsn (A) */
array[7] = 0;    /* rn (D)*/
array[8] = low_neighbor;
array[9] = high_neighbor;
array[10] = latticesize;
array[11] = 0;       /* rsn */
```

```
MPI_Barrier(MPI_COMM_WORLD);

for (x = 25; x < 26; x++) {
      x_value = ((double)x)/((double)XSIZE);
      c = x_value*latticesize;   /* total number of up sites initially in the original lattice */
      if (c % numtasks == 0) {
         *local_c = c/numtasks;   /* all processes get an equal number of down sites initially */
      } else {
          remainder = c % numtasks;
          *local_c = (c - remainder)/numtasks; /* randomly assign the leftover sites */
          if (taskid == 0) {          /* have processor 0 disperse the random sites */
                 for (t = 0; t < remainder; t++) {
                   proc = return_rand_proc(numtasks);  /* send an up site to a random process */
                  /* processor 0 accesses other processors' local c values, increments them */
                   MPI_Win_lock(MPI_LOCK_EXCLUSIVE, proc, 0, local_c_win);
                   MPI_Accumulate(&one, 1, MPI_INT, proc, 0, 1,
                          MPI_INT, MPI_SUM, local_c_win);
                   MPI_Win_unlock(proc, local_c_win);
                  }
           }
      }
      number_up_sum = 0;
      for (i = 0; i < N; i++) {
             MPI_Barrier(MPI_COMM_WORLD); /* ensure all processors are on same trial */
           c1 = *local_c;
           number_up = 0, number_down = 0;
           MCS_count = 0, localstep = 0;
           init_array(sublattice, sublatticesize);
           randomization_init(sublattice, &c1, &number_up, &number_down, sublatticesize);
           colltd_numup = 0;
           colltd_numdn = 0;  /* initial number of up/down sites (all processors combined) */
           do {
                convert_to_global_indices_and_update(sublattice, array,
                 sublattice_win, sublattice_btm, sublattice_top, &localstep);
                 number_up = 0, number_down = 0;
                 for (k = 0; k < sublatticesize; k++) { /* find number up and down in sub-lattice */
                       if (sublattice[k] == UP) number_up += 1;
                       if (sublattice[k] == DOWN) number_down += 1;
                 }
                 MPI_Allreduce(&number_up, &colltd_numup, 1, MPI_INT, MPI_SUM,
                          MPI_COMM_WORLD);
                 m = ((double)(colltd_numup - colltd_numdn))/((double)latticesize);
                 /* processor 0 records the m(t) data */
                 if (tasked == 0) fprintf(output_m, "%3f %d %d\n", m, i, latticesize);
           } while ((colltd_numup < latticesize) && (colltd_numup > 0));
           if (colltd_numup == latticesize) {
```

```
                number_up_sum = number_up_sum + 1;
            }
        }
        /* pofx reduction, print results to screen */
        if (taskid == 0) {
           printf("%f %f\n", x_value, (double)number_up_sum/(double)N);
        }
}

/*closure stuff here. */
clock_t finish = clock();

/*compute how long the simulation took in hours/minutes/seconds, print that to screen*/
double sim_time = (((double) (finish - start)) / (CLOCKS_PER_SEC));

/* write the actual time of simulation to the screen*/
printf("\nTime elapsed was: %3f s = %3f minutes, done by task #%d\n", sim_time, sim_time /
(60),taskid);

free(lattice);
MPI_Free_mem(sublattice);  /* free the memory used for the window objects */
MPI_Free_mem(local_c);
MPI_Win_free(&sublattice_win);
MPI_Win_free(&sublattice_btm);
MPI_Win_free(&sublattice_top);
MPI_Win_free(&local_c_win);

if (taskid == 0) {
        close(output);
        close(output_MCS);   /* close the file pointers for the data files */
        close(output_m);
}
MPI_Finalize();
}
/* end of main */

/* perform pair selection, map A,B,C,and D to indices, call applicable site updating function */
void convert_to_global_indices_and_update(int sublattice[], int array[], MPI_Win
sublattice_win, MPI_Win sublattice_btm, MPI_Win sublattice_top, int * localstep)
{
        int sublatticesize = array[2];
        int ss = random_at_most((sublatticesize - 1)); /* pick random site within lattice */
        int ss2 = 0, rn = 0, ln = 0;  /* initialize the values of the site pair and neighbor indices */

/* find indices of site pair and neighbors */
        int flag = global_assignments(ss, &ss2, &ln, &rn, sublatticesize);
```

```
        array[5] = ln;
        array[6] = ss;
        array[7] = rn;

        if (flag > 1) { /* updating will be done near sub-lattice boundaries, locking needed */
                update_near_neighbor_BCs(sublattice, array, sublattice_win,
                  sublattice_btm, sublattice_top, localstep);
        }
        else {  /* update a normal interior site */

                if (sublattice[ss] == sublattice[ss2]) {
                        update_near_neighbor(sublattice, array);
                }
        }
}

/* set the process topology so that periodic boundary conditions are sustained */
void neighbor_determination(int taskid, int * low_neighbor, int * high_neighbor, int numtasks)
{
        if (taskid == 0) {
                *low_neighbor = numtasks - 1;
                *high_neighbor = 1;
        } else if (taskid == (numtasks - 1)) {
                *low_neighbor =  taskid - 1;
                *high_neighbor = 0;
        } else {
                *low_neighbor = taskid - 1;
                *high_neighbor = taskid + 1;
        }
}

/* the function which does the necessary locking and updating of sites near the boundaries */
void update_near_neighbor_BCs(int sublattice[], int array[], MPI_Win sublattice_win, MPI_Win
sublattice_btm, MPI_Win sublattice_top, int * localstep)
{
/* NOTE: ANY VARIABLE ASSOCIATED WITH A WINDOW SHOULD BE LOCKED
WHENEVER IT IS PART OF AN UPDATE SELECTION, EVEN IF IT IS NOT BEING
UPDATED. WE DON'T WANT OUR ORIGINAL SITE PAIRS TO BE MODIFIED WHILE
THEY ARE MODIFYING OTHER SITES. */
        int taskid =  array[0];
        int numtasks =  array[1];
        int sublatticesize = array[2];
        int up =  array[3];
        int down = array[4];            /* diagram of the four-site panel: [ln][rsn/ss][ss2][rn] */
        int ln = array[5];              /* Alternatively, [C][A][B][D] */
        int ss = array[6];
```

```
int rn = array[7];
int low_neighbor = array[8];
int high_neighbor = array[9];
int latticesize = array[10];
int rsn = array[11];
int index_low = array[12];
int index_high = array[13];

int stateofmaxindexoflowerprocess[1];
int stateofzeroindexinthisprocess[1];
int stateofoneindexinthisprocess[1];       /* site-state tracking variables */
int stateofmaxindexinthisprocess[1];
int stateofzeroindexofhigherprocess[1];
int stateofoneindexinhigherprocess[1];

   /* note: processor X=>([…][…]) indicates that the sites (square brackets) in the
   parentheses belong to Processor X. The processor which ss/rsn/A belongs to is called
   processor I */

if (ss == 0) {  /* processor S=>([…][C])([ss/A][B][D])<=Processor I */
       /* this first lock allows for A and B to stay the same */
       MPI_Win_lock(MPI_LOCK_EXCLUSIVE, taskid, 0, sublattice_btm);
       /* this second lock allows for C to stay the same before being updated */
       MPI_Win_lock(MPI_LOCK_EXCLUSIVE, low_neighbor, 0, sublattice_top);

       if (sublattice[ss] == sublattice[ss+1]) { /* make sure UP1 is satisfied */
             *localstep += 1;  /* increment the MCS count */
             MPI_Put(&(sublattice[ss]), 1, MPI_INT, low_neighbor, 2, 1, MPI_INT,
                    sublattice_top);  /* the actual remote updating operation */
             sublattice[rn] = sublattice[ss];
       }
       MPI_Win_unlock(taskid, sublattice_btm);
       MPI_Win_unlock(low_neighbor, sublattice_top);
}
if (ss == 1) { /* processor J=>([…][…])([C][A][B][D])<=Processor I */
      /* only one lock is needed (for Processor I) */
       MPI_Win_lock(MPI_LOCK_EXCLUSIVE, taskid, 0, sublattice_btm);
       if (sublattice[ss] == sublattice[ss + 1]) {
             *localstep += 1;
             sublattice[ln] = sublattice[ss];
             MPI_Put(&(sublattice[ss]), 1, MPI_INT, taskid, 0,1, MPI_INT, sublattice_btm);
             sublattice[rn] = sublattice[ss];
       }
       MPI_Win_unlock(taskid, sublattice_btm);
}
if (ss == 2) {  /* processor J=>([…][…])([0][C][A][B][D])<=Processor I */
```

```
     /* same locking scenario as when ss = 1 */
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, taskid, 0, sublattice_btm);
      if (sublattice[ss] == sublattice[ss + 1]) {
            *localstep += 1;
            sublattice[ln] = sublattice[ss];
            MPI_Put(&(sublattice[ss]), 1, MPI_INT, taskid, 1, 1, MPI_INT, sublattice_btm);
            sublattice[rn] = sublattice[ss];
      }
      MPI_Win_unlock(taskid, sublattice_btm);
}
if (ss == (index_high - 2)) { /* processor I=>([C][A][B][D])([…][…])<=Processor J */
   /* all sites of interest are in processor I, so only one window needs to be locked */
      MPI_Win_lock(MPI_LOCK_EXCLUSIVE, taskid, 0, sublattice_top);
      if (sublattice[ss] == sublattice[ss + 1]) {
            *localstep += 1;
            sublattice[ln] = sublattice[ss];
            MPI_Put(&(sublattice[ss]), 1, MPI_INT, taskid, 2, 1, MPI_INT, sublattice_top);
            sublattice[index_high] = sublattice[ss];
      }
      MPI_Win_unlock(taskid,sublattice_top);
}
if (ss == (index_high - 1)) { /* processor I=>([C][A][B])([D][…])<=Processor J */
      /* this first lock ensures that A,B, and C remain stable */
      MPI_Win_lock(MPI_LOCK_EXCLUSIVE, taskid, 0, sublattice_top);
     /* this second lock ensures that D remains stable for the possible upcoming update */
      MPI_Win_lock(MPI_LOCK_EXCLUSIVE, high_neighbor, 0, sublattice_btm);

      if ((sublattice[ss] == sublattice[ss + 1])) {
            *localstep += 1;
            sublattice[ln] = sublattice[ss];
            MPI_Put(&(sublattice[ss]), 1, MPI_INT, high_neighbor, 0, 1, MPI_INT,
                  sublattice_btm);
      }
      MPI_Win_unlock(taskid, sublattice_top);
      MPI_Win_unlock(high_neighbor, sublattice_btm);
}
if (ss == (index_high)) {  /* processor I=>([C][A])([B][D])<=Processor J */
      /* the first lock allows for A and C to be kept stable */
      MPI_Win_lock(MPI_LOCK_EXCLUSIVE, taskid, 0, sublattice_top);
      /* the second lock allows B and D to remain stable */
      MPI_Win_lock(MPI_LOCK_EXCLUSIVE, high_neighbor, 0, sublattice_btm);
      /* check to see if site B (in processor J) agrees with A */
      MPI_Get(&(stateofzeroindexofhigherprocess[0]), 1, MPI_INT, high_neighbor, 0, 1,
            MPI_INT, sublattice_btm);
```

```
            if ((stateofzeroindexofhigherprocess[0] == sublattice[ss])) {
                *localstep += 1;
                sublattice[ln] = sublattice[ss];

                /* set D to the same state as A and B (/
                MPI_Put(&(sublattice[ss]), 1, MPI_INT, high_neighbor, 1, 1, MPI_INT,
                        sublattice_btm);
            }
            MPI_Win_unlock(taskid, sublattice_top);
            MPI_Win_unlock(high_neighbor, sublattice_btm);
        }
}

/* count the number of up and down sites after each pair selection in a sub-lattice */
void updown_ct(int * up, int * down, int state, int sublatticesize)
{
        if (state == DOWN) {
                *down += 1;
                *up = sublatticesize - *down;
        }
        if (state == UP) {
                *up += 1;
                *down = (sublatticesize) - *up;
        }
}

/* map A,B,C, D to the proper array indices (and processors) so that periodic boundary
conditions are obeyed */
int global_assignments(int ss, int * ss2, int * ln, int * rn, int sublatticesize)
{
        if ((ss > 2) && (ss < (sublatticesize - 3))) {   /* an interior site, no locking needed */
                *ss2 = ss + 1;
                *rn = ss + 2;
                *ln = ss - 1;
                return 1;
        }
        if (ss == 0) {
                *ln = sublatticesize - 1;
                *ss2 = 1;                       /* the left neighbor belongs to the "left" processor */
                *rn = 2;
                return 2;
        }
        if (ss == 1) {
                *ss2 = ss + 1;
                *rn = ss + 2;           /* normal neighbor assignment, but locking will be necessary */
                *ln = ss – 1;
```

```
            return 3;
        }
        if (ss == 2) {
            *ss2 = ss + 1;
            *rn = ss + 2;           /* same as above */
            *ln = ss - 1;
            return 4;
        }

          /* one of the site pair, and the right neighbor site belongs to the "right" processor */
        if (ss == (sublatticesize - 1)) {
            *ln = ss - 1;
            *ss2 = 0;
            *rn = 1;
            return 5;
        }
        if (ss == (sublatticesize - 2)) {
            *ln = ss - 1;
            *ss2 = ss + 1;
            *rn = 0;        /* the right neighbor belongs to the "right" processor */
            return 6;
        }
        if (ss == (sublatticesize - 3)) {
            *ss2 = ss + 1;
            *rn = ss + 2;
            *ln = ss - 1;                   /* normal neighbor assignment, but locking is needed */
            return 7;
        }
}

/* return a random integer value that corresponds to an index in the sub-lattice arrays */
long random_at_most(long max)
{
      /* adapted from an online forum */
       unsigned long num_bins = (unsigned long) max + 1;
       num_rand = (unsigned long) RAND_MAX + 1;
       bin_size = num_rand / num_bins;
       defect = num_rand % num_bins;
       long x;
       do {
            x = random();
       } while (num_rand - defect <= (unsigned long)x);
return x/bin_size;
}

/* sets all sites as down within a sub-lattice */
```

```
void init_array(int sublattice[],int sublatticesize)
{
	int taskid;
	MPI_Comm_rank(MPI_COMM_WORLD, &taskid); /* grab processor's identifier */
	int j = 0;
	for (j = 0; j < sublatticesize; j++) {
		sublattice[j] = DOWN;  /* set all sites as down */
	}
}

/* randomly distributes all of the initially-up sites within an initialized sub-lattice */
void randomization_init(int sublattice[], int * c, int * number_up, int * number_down, int
sublatticesize)
{
	int max_index = sublatticesize - 1;
	int site = 0;
	while (*c > 0) {
	site = random_at_most(max_index);  /* select a random site to set as up */
		if (sublattice[site] == DOWN) {  /* make sure that site is down */
			sublattice[site] = UP;   /* make the down site up */
			*c = *c - 1;                    /* now one less site to up */
			*number_up = *number_up + 1;       /* increase the amount of up sites by 1*/
			*number_down = sublatticesize - (*number_up);
		}
	}
}

/* function used for updating neighbors (C, D) when all sites are in the sub-lattice interior */
void update_near_neighbor(int sublattice[], int array[])
{	  /* diagram of the four-site panel: [ln][rsn/ss][ss2][rn] or [C][A][B][D] */
	int ln = array[5];
	int ss = array[6];
	int rn = array[7];

	if (sublattice[ss] == sublattice[ss + 1]) {
		sublattice[ln] = sublattice[ss];
		sublattice[rn] = sublattice[ss];     /* if UP1 is satisfied, the neighbors are updated */
	}
}

/* create file names for the data output files, open them for (appending) writing */
void create_and_open_files(FILE * output, FILE * output_MCS, FILE * output_m, char *
latticesize, char * p, char * N)
{
	char * three_strings = malloc(strlen("SM1_Data/N=") + 2*strlen(latticesize) +
	   strlen("_Runs/") + strlen(p) + strlen(N) + 3 + strlen("_MCS_930.dat"));
```

```
	char * three_strings2 = malloc(strlen("SM1_Data/N=") + 2*strlen(latticesize) +
	   strlen("_Runs/") + strlen(p) + strlen(N) + 3 + strlen("_NC_930.dat"));
	char * three_strings3 = malloc(strlen("SM1_Data/N=") + 2*strlen(latticesize) +
	   strlen("_Runs/") + strlen(p) + strlen(N) + 3 + strlen("_m(t)_930.dat"));

	concatenate_strings(three_strings, "_MCS_930.dat", latticesize, p, N);
	concatenate_strings(three_strings2, "_NC_930.dat", latticesize, p, N);
	concatenate_strings(three_strings3, "_m(t)_930.dat", latticesize, p, N);

	output_MCS = fopen(three_strings, "a");   /*naming the MCS information file*/
	output = fopen(three_strings2, "a");      /*naming the E(x,L) information file.*/
	output_m = fopen(three_strings3, "a");

	free(three_strings);
	free(three_strings2);  /* free up the memory allocated for the string buffers */
	free(three_strings3);
}



/* called to create the strings for the file names used in create_and_open_files() */
void concatenate_strings(char * string, char * message, char * latticesize, char * p, char * N)
{
	strcpy(string, "SM1_Data/N=");
	strcat(string, latticesize);
	strcat(string, "_Runs/");
	strcat(string, latticesize);
	strcat(string, "_");                    /* create the file names by combining several strings */
	strcat(string, p);
	strcat(string, "_");
	strcat(string, N);
	strcat(string, message);
}

/* returns a random integer that ranges from 0 to P – 1 */
int return_rand_proc(int numtasks)
{
	int r;
	int min = 1;
	int max = numtasks;
	r = min + (rand() % (int)(max - min + 1));
	r -= 1;
	return r;
}
```

Appendix C: The Functional Decomposition Approach LRSM Code (MPI_2.c)

MPI_2.c was written in C99, compiled with mpicc (a wrapper around gcc) and run in a Linux environment (CentOS 5.7). The instructions on how to compile and run the program with the desired parameters are listed right below the #*include* commands. As a side note, we mention that *time*(NULL) (which returns the number of seconds passed since January 1st, 1970) is used to seed the random number generator due to the fact that it has a unique value each time the program is run, therefore allowing "unique" randomness, so to speak.

```
#include <mpi.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>
#include <sys/time.h>
#include <sys/resource.h>


/* Here is what the compiling command should be: */
/* mpicc –o MPI_2 MPI_2.c */
/* Here is what the command line implementation should look like: */
/* mpiexec -n (number of desired processors) ./MPI_2 TOTALSIZE P_VALUE
NUM_OF_RUNS */
/*              [0]   [1]     [2]     [3]       */

#define XSIZE         100
#define UP            1
#define DOWN           0

/* lattice initialized to have only down sites */
void init_array(int lattice[],int latticesize);

/* lattice's up sites are randomly distributed */
void randomization_init(int lattice[], int * c, int * number_up, int * number_down, int
latticesize);

/* pick a random site within the sub-lattice */
int random_num_interval (int latticesize);
```

```
void neighbor_find_update(int lattice[], int array[], int * number_up, int * number_down, double
p);

void update_neighbors(int lattice[], int array[], double p_cA, double p_cB, double p);

/* establish the indices of the site pair and the immediate neighbors */
int global_assignments(int global_ss, int * global_ss2, int * global_ln, int * global_rn, int
latticesize);

/* create the files for data writing, give them the proper file names */
void create_and_open_files(FILE * output, FILE * output_MCS, FILE * output_m, char *
latticesize, char * p, char * N);

/* helper function used to establish files names */
void concatenate_strings(char * string, char * message, char * latticesize, char * p, char * N);

/* random number generator, used to pick random sites in lattices */
long random_at_most(long max);

void main (int argc, char *argv[])
{
if (argc != 4 ) { /* check that program is being properly used */
      printf("You forgot to include one or more arguments.\n");
}

struct rlimit rlim = { RLIM_INFINITY, RLIM_INFINITY };
if ( setrlimit(RLIMIT_STACK, &rlim) == -1 ) {                    /* increase stack size */
      perror("setrlimit error");
      exit(1);
}

int random_time = time(NULL);  /*seed that random integer with the current time, etc.*/
srand(random_time);    /*continuation of seeding the pseudorandom number generator*/
clock_t start;
char   buffer[1024];
int     numtasks;
int     reminder;
int     latticesize;
double p;
int  N;
int lattice;
int taskid;
int i;
int     j;
int     x;
int     c;                    /* various loop tracker variables */
```

```
int     t;
int     k;
int     one = 1;
int     proc;
int number_up;
int number_down;
int     number_up_sum;
int     localstep;
int     MCS_count;
int     rsn;
int    c1;
int     remainder = 0;
double x_value;
double m;

FILE * output = 0;              /*output file pointer for all E(x,L) information*/
FILE * output_MCS = 0;          /*output file pointer for all MCS information*/
FILE * output_m = 0;            /*output file pointer for all m(t) information */

int    array[14] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0};   /* various numbers that each process should
know: e.g., taskid, latticesize, etc.*/

start = clock();  /* grab simulation start time (want to include MPI initializations) */

/***** MPI Initializations *****/
MPI_Init(&argc, &argv);                 /* initialize the MPI context */
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);         /* use MPI_COMM_WORLD to
determine number of tasks (processors) */
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);

MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_RETURN);

latticesize = atoi(argv[1]);
p = atof(argv[2]);
N = atoi(argv[3]);
int orig_N = N;
fprintf(stdout," latticesize = %d, p = %f, orig_N = %d\n", latticesize, p, orig_N);

if((N % numtasks) == 0) {
          N = N/numtasks;   /* all processors get an equal number of iterations */
}
if ((N % numtasks) != 0) {
  remainder = N % numtasks;    /* determine number of leftover iterations */
  N = (N - remainder)/numtasks; /* all processors get an equal base number of iterations */
  if (taskid == 0) {
     N = N + remainder;   /* process 0 takes the hit and gets the extra trials */
```

```
  }
}  /* some c_locals should be higher than they were before at this point */
MPI_Barrier(MPI_COMM_WORLD); /* synchronize, make sure each processor has the right
number of trials */
lattice = (int *)malloc(latticesize*sizeof(int));

if (taskid == 0) {/* processor 0 sets the data file names, opens them for writing (appending) */
      sprintf(buffer, "N%d_%f_%d_mt_MAR30.dat", latticesize, p, orig_N);
      output_m = fopen(buffer, "a");
      sprintf(buffer, "N%d_%f_%d_MCS_MAR30.dat", latticesize, p, orig_N);
      output_MCS = fopen(buffer, "a");
}

array[0] = taskid;
array[1] = numtasks;
array[3] = 0;   /* number_up */
array[4] = 0;    /* number_down */
array[5] = 0;    /* ln */
array[6] = 0;    /* ss (or rsn) */
array[7] = 0;    /* rn */
array[10] = latticesize;
array[11] = 0;       /* rsn */

for (x = 0; x < (XSIZE+1); x++) { /* the x-values for-loop */
      x_value = ((double)x)/((double)XSIZE);
      c = x_value*latticesize;   /* the total number of initial up sites */
      number_up_sum = 0; /* number of trials that have resulted in a unanimously up lattice */
      for (i = 0; i < N; i++) {  /* the trials for-loop */
            number_up = 0;
            number_down = 0;
            MCS_count = 0;
            localstep = 0;
            c1 = c;
            init_array(lattice, latticesize);    /* initialize the lattice for each trial */

           /* distribute the initial up sites randomly */
            randomization_init(lattice, &c1, &number_up, &number_down, latticesize);


            /* the consensus while-loop */
            do {
                  neighbor_find_update(lattice, array, &number_up, &number_down, p);
                  localstep += 1;
                  number_up = 0, number_down = 0;
                  for (k = 0; k < latticesize; k++) {
                        if (lattice[k] == UP) number_up += 1; /* count number of up-sites */
```

```
            }
            number_down = latticesize - number_up;

            if ((taskid == 0) && ((localstep % latticesize) == 0)) {
              /* processor 0 gathers the magnetization information at every full MCS */
                m = ((double)(number_up - number_down))/((double)latticesize);
                if (output_m == NULL) fprintf(stdout,"output_m file pointer NULL\n");
                MCS_count = localstep / latticesize;
                fprintf(output_m, "%d %f %d %f %d\n", taskid, x_value, i, m, MCS_count);
            }

        } while ((number_up < latticesize) && (number_up > 0));

        if ((taskid == 0) && (x_value == 0.5)) {
              /* processor 0 gathers the MCS information for all trials performed at x = 0.5 */
            if (output_MCS == NULL) fprintf(stdout,"output_MCS file pointer NULL\n");
            fprintf(output_MCS,"%f %d %d",x_value, i, MCS_count);
        }
        if (number_up == latticesize) {
            number_up_sum += 1;  /* increment "U" (see Eq. (1)) by 1 */
        }
    }
    fprintf(stdout, "%d %f %d %d\n", taskid, x_value, number_up_sum, N);  /* print out the
E(x) data */
}

/* closure stuff here */
clock_t finish = clock();

/* compute how long the simulation took in hours/minutes/seconds, print that to screen */
double simtime = (((double) (finish - start)) / (CLOCKS_PER_SEC));
fprintf(stdout, "simtime is %f in process %d\n", simtime, taskid);

free(lattice);   /* free the array created to represent the lattices under study */

if (taskid == 0) {                  /* processor 0 closes the file pointers */
    close(output_MCS);
    close(output_m);
}

MPI_Finalize();
}
/* end of main */

/* perform the pair selection, determine if UP1 is satisfied, call the site updating function */
void neighbor_find_update(int lattice[], int array[], int * number_up, int * number_down,
double p)
```

```
{
    int latticesize = array[10];
    int ss = random_at_most((latticesize - 1)); /* pick random site within lattice */
    int ss2 = 0, rn = 0, ln = 0; /* initialize values for the indices of pair sites/ neighbor sites */
    double p_cA = 0;
    double p_cB = 0;

/* probabilities for long-range sites to be updated */
    p_cA = ((double)rand()) / ((double)RAND_MAX);
    p_cB = ((double)rand()) / ((double)RAND_MAX);

 /* determine what the four sites of interest are. */
    global_assignments(ss, &ss2, &ln, &rn,latticesize);

    array[5] = ln;
    array[6] = ss;
    array[7] = rn;

    if (lattice[ss] == lattice[ss2]) { /* if UP1 is satisfied, the pair can update others */
        update_neighbors(lattice, array, p_cA, p_cB, p);
    }
}

/* map A,B,C,D to the proper array indices so that periodic boundary conditions are obeyed */
int global_assignments(int ss, int * ss2, int * ln, int * rn, int latticesize)
{
    if (ss == 0) { /* neighbor configuration [ss ss2 rn x x x x x x x x x ln] */
          *ln = latticesize - 1;
          *ss2 = 1;
          *rn = 2;
    }
    else if (ss == (latticesize - 1)) { /* neighbor configuration [ss2 rn x x x x x x x x x ln ss] */
*/
          *ln = ss - 1;
          *ss2 = 0;
          *rn = 1;
    }
    else if (ss == (latticesize - 2)) { /* neighbor configuration [rn x x x x x x x x x ln ss ss2] */
          *ln = ss - 1;
          *ss2 = ss + 1;
          *rn = 0;
    } else {                /* standard neighbor configuration [x x x x ln rsn ss2 rn x x x x x] */
        *ss2 = ss + 1;
        *ln = ss - 1;
        *rn = ss + 2;
    }
```

```
}


/* return a random integer value that corresponds to an index in the sub-lattice arrays */
long random_at_most(long max)

{
     /* adapted from an online forum */
      unsigned long num_bins = (unsigned long) max + 1;
      num_rand = (unsigned long) RAND_MAX + 1;
      bin_size = num_rand / num_bins;
      defect = num_rand % num_bins;
      long x;
      do {
            x = random();
      } while (num_rand - defect <= (unsigned long)x);
return x/bin_size;
}


/* set all sites within the lattice as having the down opinion */
void init_array(int lattice[],int latticesize)
{
      int taskid;
      MPI_Comm_rank(MPI_COMM_WORLD, &taskid); /* grab processor's identifier */
      int j = 0;
      for (j = 0; j < latticesize; j++) {
            lattice[j] = DOWN;  /* set all sites as down */
      }
}

/* randomly places the initially-up sites within the lattice for the beginning of a trial */
void randomization_init(int lattice[], int * c, int * number_up, int * number_down, int latticesize)
{
      int max_index = latticesize - 1;
      int site = 0;
      while (*c > 0) {
      site = random_at_most(max_index);  /* select a random site to set as up */
            if (lattice[site] == DOWN) {  /* make sure that site is down */
                  lattice[site] = UP;   /* make the down site up */
                  *c = *c - 1;                        /* now one less site to up */
                  *number_up = *number_up + 1;       /* increase the amount of up sites by 1*/
                  *number_down = latticesize - (*number_up);
            }
      }
}
```

```
/* the function which actually updates sites (UP1 is satisfied when this is called) */
void update_neighbors(int lattice[], int array[], double p_cA, double p_cB, double p)
{       /* diagram of the four-site panel: [ln][rsn/ss][ss2][rn] or [C][A][B][D] */
      int ln = array[5];
      int ss = array[6];
      int rn = array[7];
      int latticesize = array[10];
      int lr_A = rand() % latticesize;       /* select random long-range sites */
      int lr_B = rand() % latticesize;
      if ((p_cA < p) && (p_cB < p)){ /* both updates are long range */
            lattice[lr_A] = lattice[ss];
            lattice[lr_B] = lattice[ss];
      }
      if ((p_cA > p) && (p_cB > p)) { /* both updates are short range */
            lattice[ln] = lattice[ss];
            lattice[rn] = lattice[ss];
      }
      if ((p_cA < p) && (p_cB > p)) {
/* left site (A) updates long-range neighbor, right site (B) updates immediate neighbor */
            lattice[lr_B] = lattice[ss];
            lattice[rn] = lattice[ss];
      }
      if ((p_cA > p) && (p_cB < p)) {
/* A updates immediate neighbor, B updates long-range neighbor */
            lattice[ln] = lattice[ss];
            lattice[lr_B] = lattice[ss];
      }
}

/* create file names for the data output files, open them for (appending) writing */
void create_and_open_files(FILE * output, FILE * output_MCS, FILE * output_m, char *
latticesize, char * p, char * N)
{
      char * three_strings = malloc(strlen("SM1_Data/N=") + 2*strlen(latticesize) +
         strlen("_Runs/") + strlen(p) + strlen(N) + 3 + strlen("_MCS_930.dat"));
      char * three_strings2 = malloc(strlen("SM1_Data/N=") + 2*strlen(latticesize) +
         strlen("_Runs/") + strlen(p) + strlen(N) + 3 + strlen("_NC_930.dat"));
      char * three_strings3 = malloc(strlen("SM1_Data/N=") + 2*strlen(latticesize) +
         strlen("_Runs/") + strlen(p) + strlen(N) + 3 + strlen("_m(t)_930.dat"));

      concatenate_strings(three_strings, "_MCS_930.dat", latticesize, p, N);
      concatenate_strings(three_strings2, "_NC_930.dat", latticesize, p, N);
      concatenate_strings(three_strings3, "_m(t)_930.dat", latticesize, p, N);

      output_MCS = fopen(three_strings, "a");   /*naming the MCS information file*/
      output = fopen(three_strings2, "a");      /*naming the E(x,L) information file.*/
```

```
	output_m = fopen(three_strings3, "a");

	free(three_strings);
	free(three_strings2);  /* free up the memory allocated for the string buffers */
	free(three_strings3);
}

/* called to create the strings for the file names used in create_and_open_files() */
void concatenate_strings(char * string, char * message, char * latticesize, char * p, char * N)
{
	strcpy(string, "SM1_Data/N=");
	strcat(string, latticesize);
	strcat(string, "_Runs/");
	strcat(string, latticesize);
	strcat(string, "_");                    /* create the file names by combining several strings */
	strcat(string, p);
	strcat(string, "_");
	strcat(string, N);
	strcat(string, message);
}
```

Appendix D: The Critical Exponent Finding Code

The CEF code was written in MATLAB R2014a and run in a Windows 7 environment. The files shown below (seven, one for each of the system sizes) are evaluated for the mean-field case. The question set in the middle of the code is used for several reasons:

- Channel the MATLAB code to properly evaluate the exit probability data for the two raw data formats we used in the study
- Choosing the following:
    - An x-interval over which to do the least-squares fitting
    - The range of λ and ν (and the increment value for each variable)
    - Several graphical options

```
% Using the p = 1 files
% file1_name = '100_1.000000_100000_SM1_NC_Jan_13.dat';
% file2_name = '500_1_50000_NC_104.dat';   %   2nd, N = 500 (NIY)
```

```
% file3_name = '1000_1.000000_80000_SM1_NC_Jan_13.dat';   %   3rd  N = 1000
% file4_name = '5000_1.000000_60000_SM1_NC_Jan_14.dat';  %    4th, N = 5000
% file5_name = '10000_1_30000_NC_105.dat';  %   5th, N = 10000
% file6_name = '50000_1.000000_5000_SM1_NC_Jan_28.dat';  %    6th, N = 50000
% file7_name = '100000_1.000000_1000_SM1_NC_Jan_28.dat'; %  7th, N = 100,000

%Declaring Constants here
N_1 = 100; N_2 = 500; N_3 = 1000; N_4 = 5000; N_5 = 10000; N_6 = 50000; N_7 = 100000;

%want to ensure that each file is matched with the appropriate system size!!!!
SYS_SIZE_ARRAY = [N_1; N_2; N_3; N_4; N_5; N_6; N_7]; %N_7];

X_CRIT = 0.5;            % constant value in the theoretical tanh function
delimiter_char = ' ';   % delimiting character to distinguish columns in the imported data.

file1 = importdata(file1_name, delimiter_char);    % N_1 = 100
file2 = importdata(file2_name, delimiter_char);    % N_2 = 500
file3 = importdata(file3_name, delimiter_char);    % N_3 = 1000
file4 = importdata(file4_name, delimiter_char);    % N_4 = 5000
file5 = importdata(file5_name, delimiter_char);    % N_5 = 10000
file6 = importdata(file6_name, delimiter_char);    % N_6 = 50000
file7 = importdata(file7_name, delimiter_char);    % N_7 = 100000


question = 'For a conventional data set (x goes from 0 to 1), press 1. For a non-conventional data
set (e.g. x = 0.25 to 0.75), press 0.\n';
answer = input(question);

% these remaining questions are asked by default
   question = 'What is the delta_x value?\n';
   delta_x = input(question);
   question = 'What is the top value (e.g. x = 0.75)?\n';
   top_num = input(question);
   question = 'What is the bottom value (e.g. x = 0.25)?\n';
   bottom_num = input(question);

   if (answer == 1)   % conventional data set
   NUM_OF_X_INTERVALS = 1 + ((top_num - bottom_num)/(delta_x));
   end
   if (answer == 0)   % non-conventional data set
   NUM_OF_X_INTERVALS = ((top_num - bottom_num)/(delta_x));
   end

   question = 'What is the lower limit on lambda?\n';
   lower_lambda = input(question);
```

```
   question = 'What is the upper limit on lambda?\n';
   upper_lambda = input(question);

   question = 'What is the lower limit on nu?\n';
   lower_nu = input(question);

   question = 'What is the upper limit on nu?\n';
   upper_nu = input(question);

   question = 'What is the number of indices?\n';
   NUM_OF_INDICES = input(question);

   question = 'What is the graph tick mark increment value (e.g. 100)?\n';
   TICK_INCREMENT = input(question);

   question = 'What is our starting index (1 to include N = 100, 2 to exclude N = 100)\n';
   filestart_index = input(question);

   question = 'What is our ending index (e.g., 7 would mean going all the way to N = 100000)';
   fileend_index = input(question);

   question = 'Will we be analyzing Joe data (press 1) or Tom Data? (press 0)\n';  % Joe = 1,
Tom = 0.
   person_answer = input(question);

question = 'Are we doing the Integer format (1) or the doing the decimal format (0) for exit
probability?\n';
num_answer = input(question);


% Divide lambda-range into NUM_OF_INDICES values (more nu-values gives better precision
lambda = linspace(lower_lambda,upper_lambda,NUM_OF_INDICES);

% Divide nu-range into NUM_OF_INDICES values (more nu-values gives better precision
nu = linspace(lower_nu,upper_nu,NUM_OF_INDICES);

% make sure that we are evaluating the simulation and theoretical functions at the same x-values
x_array = linspace(bottom_num,top_num,NUM_OF_X_INTERVALS);

lambda_index = 0;   % index to go through the array of lambda values..
nu_index = 0;       % index to go through the array of nu values
x_index = 0;        % index to go through the x-values (for the E(x,L) data.
lambda_value = 0.0;  %specific value of lambda at a particular index
nu_value = 0.0;      % specific value of nu at a particular index

E_simulation = 0.0;           % value of E(x,L) as generated by simulations
```

```
E_theory = 0.0;            %value of the Exit probability predicted by tanh function
tanh_component = 0;        % component of the tanh value calculation being initialized to zero.

error_max = 10000.0;       % default maximum error value.

%these two variables will be stored into memory whenever we find a lambda-nu pair that does
well with all of the seven datasets.
best_lambda = 0.0;
best_nu = 0.0;

% Variables which will be used for the heatmap applications
error_sum = 0;
lambda_nu_error_sum = 0;

% create the matrix which holds the cumulative error across all files, for each lambda-nu pair
M_error = zeros(NUM_OF_INDICES);

%*****************************************************

% The General Data Analysis Loop
for lambda_index=1:NUM_OF_INDICES                %BEGIN LAMBDA_LOOP
   lambda_value = lambda(lambda_index);       % grab the instantaneous value of lambda.
      for nu_index=1:NUM_OF_INDICES           %BEGIN NU_LOOP
         nu_value = nu(nu_index);             % grab the instantaneous value of nu.
            lambda_nu_error_sum = 0;    % reinitialize lambda_nu_error_sum for new pair
                     % BEGIN_FILE_LOOP (loop through the seven different files
            for file_index=filestart_index:fileend_index
               error_sum = 0;   % reinitialize file error for each new file
                  for x_index=1:NUM_OF_X_INTERVALS   % BEGIN Go from 0 to 1
                     %sort out what simulation value to grab here
                     if file_index == 1
                       if person_answer == 1
                          E_simulation = file1(x_index,2);
                       end
                       if person_answer == 0
                           E_simulation = file1(x_index,2);
                           if num_answer == 1
                              E_simulation = file1(x_index,3)/file1(x_index,2);
                           end
                       end
                     end
                     if file_index == 2
                        if person_answer == 1
                          E_simulation = file2(x_index,2);
                       end
                       if person_answer == 0
```

```
        E_simulation = file2(x_index,2);
        if num_answer == 1
           E_simulation = file2(x_index,3)/file2(x_index,2);
        end
    end
end
if file_index == 3
    if person_answer == 1
       E_simulation = file3(x_index,2);
    end
    if person_answer == 0
        E_simulation = file3(x_index,2);
        if num_answer == 1
           E_simulation = file3(x_index,3)/file3(x_index,2);
        end
    end
end
if file_index == 4
    if person_answer == 1
       E_simulation = file4(x_index,2);
    end
    if person_answer == 0
        E_simulation = file4(x_index,2);
        if num_answer == 1
           E_simulation = file4(x_index,3)/file4(x_index,2);
        end
    end
end
if file_index == 5
    if person_answer == 1
       E_simulation = file5(x_index,2);
    end
    if person_answer == 0
        E_simulation = file5(x_index,2);
        if num_answer == 1
           E_simulation = file5(x_index,3)/file5(x_index,2);
        end
    end
end
if file_index == 6
    if person_answer == 1
       E_simulation = file6(x_index,2);
    end
    if person_answer == 0
        E_simulation = file6(x_index,2);
        if num_answer == 1
```

```
                    E_simulation = file6(x_index,3)/file6(x_index,2);
                end
            end
        end
        if file_index == 7
            if person_answer == 1
                E_simulation = file7(x_index,2);
            end
            if person_answer == 0
                E_simulation = file7(x_index,2);
                if num_answer == 1
                    E_simulation = file7(x_index,3)/file7(x_index,2);
                end
            end
        end

            x_value = x_array(x_index);   %grab an instantaneous value of x

                % compute the tanh( ) function evaluation (in several steps)
            tanh_1 = (lambda_value/X_CRIT)*(x_value - X_CRIT);
            tanh_2 = SYS_SIZE_ARRAY(file_index);
            tanh_component = tanh(tanh_1*((tanh_2)^(nu_value^(-1))));
            E_theory = X_CRIT*(1+tanh_component);

% find difference between data and theory, add it to the error (for a particular file/system size)
            error_sum = error_sum + [E_simulation - E_theory]*[E_simulation - E_theory];
        end                 %END GO FROM 0 to 1
    % increment cumulative error (boosted by each new file/system size)
        lambda_nu_error_sum = lambda_nu_error_sum + error_sum;
    end                 %END FILE_LOOP
        if lambda_nu_error_sum < error_max
            error_max = lambda_nu_error_sum;
            best_lambda = lambda_value;             % determine best new exponents
            best_nu = nu_value;
        end
        % Assign cumulative error to lambda-nu pair, store in the error matrix
        M_error(lambda_index,nu_index) = lambda_nu_error_sum;
    end                 %END NU_LOOP
end                     %END LAMBDA_LOOP

% all pairs tried, now report the best lambda, nu values
fprintf('The best lambda is %f and the best nu is %f with a cumulative error of error_max =
    %f\n\n', best_lambda,best_nu,error_max);

M_error_inv = M_error.^(-1);  % invert the error matrix, so now it is a best fit matrix
figure;         % create figure object
```

```
mesh(M_error_inv)        % create the heatmap (actually a 3d surface)

% set up the axes' properties
xlabel('\nu')
set(gca, 'XLim', [0 NUM_OF_INDICES])
set(gca, 'XTick', [0:TICK_INCREMENT:NUM_OF_INDICES])
set(gca,'XTickLabel',[lower_nu:((upper_nu -
lower_nu)*TICK_INCREMENT/NUM_OF_INDICES):upper_nu]) % x is nu

ylabel('\lambda')
set(gca, 'YLim', [0 NUM_OF_INDICES])
set(gca, 'YTick', [0:TICK_INCREMENT:NUM_OF_INDICES])
set(gca,'YTickLabel',[lower_lambda:((upper_lambda -
lower_lambda)*TICK_INCREMENT/NUM_OF_INDICES):upper_lambda])  %y is lambda

zlabel('Error(\nu,\lambda)')
```

Author's Biography

Joseph was born in Bangor, Maine, on March 12, 1994, and has called Etna, Maine home for his whole life. After attending St. Agnes Parochial School in Pittsfield from preschool to the 8th grade, Joseph graduated from Nokomis Regional High (located in the nearby town of Newport) in June of 2012, having established himself in the top 10 percent of his class.

Originally intending to enter college as a physics major that fall, he tried Engineering Physics with a concentration in Electrical and Computer Engineering at the suggestion of his parents; he has come to love the major over the course of his four years at the University of Maine, and has performed well, earning a membership in Sigma Pi Sigma. In May 2016 Joseph graduates from the University and he hopes to find a research position at a national laboratory over the next year to build his research skills in the field of quantum computation. After finishing that, his dream is to enter graduate school to pursue a Ph.D. in physics, focusing on quantum computation.